Macro Assembler Reference

IBM Personal Computer XENIX™ Software Development System Version 2.00

Programming Family

NOT
FOR SALE
For Demonstration
or Educational
Purposes Only
D 284556

IBM

Personal
Computer
Software

XENIX is a trademark of
Microsoft Corporation.

59X8654

Macro Assembler Reference

IBM Personal Computer XENIX™ Software Development System Version 2.00

Programming Family

Personal
Computer
Software

XENIX is a trademark of
Microsoft Corporation.

Second Edition (April 1986)

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication. This edition applies to Version 2.00 of the IBM Personal Computer XENIX Operating System, and to all subsequent releases until otherwise indicated in new editions or technical newsletters.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Portions of the code and documentation described in this book were developed at the Electrical Engineering and Computer Science Department at the Berkeley campus of the University of California under the auspices of the Regents of the University of California.

© Copyright International Business Machines Corporation 1984, 1986
© Portions copyright by Microsoft Corporation 1983, 1984, 1986, all rights reserved.
© Portions copyright by AT&T Bell Laboratories 1983, 1984, 1986, all rights reserved.

About This Book

This book describes the usage and input syntax of the IBM Personal Computer XENIX Macro Assembler. This assembler produces relocatable object modules from 8086, 80186, or 80286 assembly language source files. You can link the relocatable object modules to create executable programs that can run under the XENIX Operating System.

Running on 80286-based IBM Personal Computers, the Macro Assembler can assemble the instructions of the 8086, 80186, and 80286 microprocessors, and the 8087 and 80287 floating-point coprocessors. It has a powerful set of assembly language directives. These directives give you control of the segmented architecture of the 8086, 80186, and 80286 microprocessors. The Macro Assembler instruction syntax allows a variety of operand data types, including the following:

- Integers
- Strings
- Packed decimals
- Floating-point numbers
- Structures
- Records.

The Macro Assembler has a full set of macro directives that let you create and use macros in a source file. The directives instruct the assembler to repeat common blocks of statements or replace macro names with the block of statements they represent. The Macro Assembler also has conditional directives that let you exclude portions of a source file from assembly or include additional program statements by defining a symbol.

The Macro Assembler carries out strict syntax checking of all instruction statements, including strong typing for memory operands. Unlike some other assemblers, this assembler detects questionable operand use that can lead to errors or results other than those you want.

The Macro Assembler produces object modules that are compatible with object modules that high-level language compilers create. Thus, you can make complete programs by combining IBM Personal Computer XENIX Macro Assembler object modules with object modules that the IBM Personal Computer XENIX C Language compiler creates.

Background Knowledge

This book does not teach assembly language programming, nor does it give a detailed description of 8086, 80186, and 80286 instructions. You need to know the function and operation of the instructions in the instruction sets of the 8086/80186/80286 family of microprocessors.

You can link the object module produced by the Macro Assembler with other object modules to produce a load module for PC XENIX. (See **cc** and **ld** in *XENIX Commands Reference* for more information.)

For PC XENIX, the **as** command assembles nonprotected 80286 and 80287 instructions by default. Since all of the 8086 and 8087 instructions are included in the nonprotected 80286 and 80287 instruction set, 8086 and 8087 instructions are also recognized by default. Therefore, the object modules can be used with the DOS cross-development facility to produce an executable program that can run under DOS. (See **cc** and **dosld** in *XENIX Commands Reference* for more information.)

For your convenience, Appendix A, "Instruction Summary" contains a list of the instruction names and syntax for all the previously named processors.

Notational Conventions

This book uses the following notational conventions:

[ ]	Brackets indicate an optional command argument. If you do not use the optional item, the program selects a default action.
bold	**Bold** type indicates **commands**, **options**, **file names**, **programs**, and **functions**. You must enter these **bold** characters exactly as shown. We also use **bold** type to identify the exact wording of text on the screen.
italics	*Italics* indicate variables for *placeholders*, *arguments*, and *filenames*. When you enter a command, replace all variables with the appropriate file name, number, or option. We also use *italics* the first time a glossary term appears in this book. The glossary is in *XENIX Glossary and Master Index*.
. . .	Ellipses after an argument indicate that you can repeat the preceding item any number of times.

,,,	Commas indicate that you can repeat the preceding item any number of times as long as you separate the items with a comma.
¦	Vertical bars indicate that you can use only one of the separated items. You must make a choice between the items.

All directives, operands, and other important syntax can be in uppercase or lowercase, unless otherwise noted.

Throughout the book, we use blue to highlight examples and syntax. When we show syntax, replace only the characters in italics. Type all other characters exactly as shown.

Contents

Chapter 1. The Assembly Command 1-1
Starting and Using the Assembler 1-1
How to Begin 1-1
Assembling a Source File 1-5
Using the Options 1-6
Alphabetical Ordering for Segments 1-7
Output a Pass 1 Listing 1-7
Define Symbols 1-7
Include File Path Names 1-8
Output Listing File 1-8
Preserve Case Sensitivity in Names 1-9
Map All Symbols to Uppercase 1-9
Preserve Case Sensitivity in Externals 1-9
Suppress Symbol Table Information 1-10
Real Floating-Point Instructions 1-11
Output Assembly Statistics 1-11
List Errors to Standard Error Channel 1-12
List False Conditionals 1-12
Reading the Assembly Listing 1-13
Reading Program Code 1-13
Reading a Macro Table 1-15
Reading a Structure and Record Table 1-15
Reading a Segment and Group Table 1-17
Reading a Symbol Table 1-18
Reading a Pass 1 Listing 1-20

Chapter 2. Elements of the Assembler 2-1
Characters, Numbers, and Names 2-1
Character Set 2-1
Integers 2-2
Real Numbers 2-3
Encoded Real Numbers 2-3
Packed Decimal Numbers 2-4
Character and String Constants 2-4
Names 2-5
Reserved Names 2-6

Statements and Comments 2-7
  Statements 2-7
  Comments 2-8
  COMMENT Directive 2-8

Chapter 3. Program Structure 3-1
Introduction 3-1
Source Files 3-1
Instruction Set Directives 3-3
SEGMENT and ENDS Directives 3-4
  Program Loading Options 3-6
END Directive 3-7
GROUP Directive 3-8
ASSUME Directive 3-9
ORG Directive 3-10
EVEN Directive 3-10
PROC and ENDP Directives 3-11

Chapter 4. Types and Declarations 4-1
Label Declarations 4-1
  Near Label Declarations 4-1
  Procedure Labels 4-2
Data Declarations 4-2
  DB Directive 4-3
  DW Directive 4-4
  DD Directive 4-5
  DQ Directive 4-6
  DT Directive 4-7
  DUP Operator 4-8
Symbol Declarations 4-8
  = Directive 4-9
  EQU Directive 4-9
  LABEL Directive 4-10
Type Declarations 4-11
  STRUC and ENDS Directives 4-11
  RECORD Directive 4-13
Structure and Record Declarations 4-14
  Structure Declarations 4-14
  Record Declarations 4-15

Chapter 5. Operands and Expressions 5-1
Introduction 5-1
Operands 5-1
  Constant Operands 5-2
  Direct Memory Operands 5-2
  Relocatable Operands 5-2
  Location Counter Operands 5-3
  Register Operands 5-3
  Based Operands 5-5
  Indexed Operands 5-5
  Based Indexed Operands 5-6
  Structure Operands 5-7
  Record Operands 5-8
  Record Field Operands 5-8
Expressions 5-9
  Arithmetic Operators 5-10
  SHR and SHL Operators 5-11
  Relational Operators 5-11
  Bitwise Operators 5-12
  Index Operator 5-13
  PTR Operator 5-13
  Segment Override Operator 5-14
  SHORT Operator 5-15
  THIS Operator 5-15
  HIGH and LOW Operators 5-16
  SEG Operator 5-16
  OFFSET Operator 5-16
  TYPE Operator 5-17
  .TYPE Operator 5-17
  LENGTH Operator 5-18
  SIZE Operator 5-19
  WIDTH Operator 5-19
  MASK Operator 5-20
  Expression Evaluation and Precedence 5-20
Forward References 5-21
Strong Typing for Memory Operands 5-23

Chapter 6. Global Declarations 6-1
Introduction 6-1
PUBLIC Directive 6-2
EXTRN Directive 6-2
Program Example 6-3

Chapter 7. Conditional Assembly 7-1
Introduction 7-1
IF and IFE Directives 7-2
IF1 and IF2 Directives 7-2
IFDEF and IFNDEF Directives 7-3
IFB and IFNB Directives 7-3
IFIDN and IFDIF Directives 7-4

Chapter 8. Macro Directives 8-1
Introduction 8-1
MACRO and ENDM Directives 8-2
Macro Calls 8-4
LOCAL Directive 8-5
PURGE Directive 8-6
REPT and ENDM Directives 8-7
IRP and ENDM Directives 8-7
IRPC and ENDM Directives 8-9
EXITM Directive 8-9
Substitute Operator 8-10
Literal Text Operator 8-11
Literal Character Operator 8-12
Expression Operator 8-12
Macro Comment 8-13

Chapter 9. File Control 9-1
Introduction 9-1
INCLUDE Directive 9-2
.RADIX Directive 9-2
%OUT Directive 9-3
NAME Directive 9-4
TITLE Directive 9-4
SUBTITLE Directive 9-5
PAGE Directive 9-5
.LIST and .XLIST Directives 9-6
.SFCOND, .LFCOND, and .TFCOND Directives 9-7
.LALL, .XALL, and .SALL Directives 9-8
.CREF and .XCREF Directives 9-8

Appendix A. Instruction Summary A-1
8086 Instructions A-2
8087 Instructions A-9
80186 Instructions A-13
80286 Nonprotected Instructions A-14
80286 Protected Instructions A-15
80287 Instructions A-16

Appendix B. Directive Summary B-1
Introduction B-1
Directive Syntax and Function B-2

Appendix C. Segment Names For High-Level Languages C-1
  TEXT Segments C-2
  DATA Segments - Near C-3
  DATA Segments - Far C-5
  BSS Segments C-6
  CONST Segments C-7

Appendix D. Macro Assembler and Linker Error Messages D-1
Macro Assembler Messages D-1
Assembler Errors D-2
Linker Messages D-15

Index X-1

Ordering Additional Copies of This Book

To order additional copies of this publication (*IBM Personal Computer XENIX Macro Assembler Reference*) use either of the following sources:

- To order from your IBM representative, use Order Number SV21-8079.
- To order from your IBM dealer, use Part Number 59x9940.

This publication includes the following:

- *IBM Personal Computer XENIX C Library Guide and Compiler Reference*
- *IBM Personal Computer XENIX Macro Assembler Reference*
- Binder
- Slipcase.

Chapter 1. The Assembly Command

This chapter describes how to use the **as** command to assemble programs written using the IBM Personal Computer XENIX Macro Assembler language.

Starting and Using the Assembler

This section explains how to start and use the **as** command to assemble your program source files.

How to Begin

You begin by creating an assembly language source file with a text editor. Then, you assemble the source file using the **as** command.

When writing assembly language routines for PC XENIX, it is important to always remember that PC XENIX is a C-based operating system. This means that the main programs are really just procedures that obey the C conventions for the particular memory model selected:

- Small with combined code and data
- Small with separate code and data
- Medium
- Large
- Huge.

For easier readability, assembly language programs should be organized as follows:

1. macros, strucs, records, and nondata dependent equates
2. data segments and declarations
3. code segment(s)
4. end statement

In general, you can avoid most startup problems with the Macro Assembler by following a set of conventions such as those described here for PC XENIX. The recommended approach is to construct as small an assembly language routine as possible and try to interface it with C. Because the C compiler can generate assembly language files, start by writing a small C function and compile it with the C compiler using the **-S** option.

For example, create the following C program using one of the PC XENIX editors:

```
main ( )
{
   printf ("Hello\n");
}
```

Save this C program in a file called **example.c**.

Enter **cc -S example.c** to generate the assembly code in the file called **example.s**.

Examine the generated code and format in the **example.s** file using a PC XENIX command, such as **more** (type, **more example.s**).

In PC XENIX and C, names are case sensitive. As a general rule for working with assembly language:

- Segment and group names should be in uppercase.
- Class names should be in uppercase.
- Public and external names should be in the case you desire (usually lowercase).

All public and external names that can be referenced from C have a leading underscore (_) in assembly language. Any public names without one underscore and only one in front of them may conflict with "private" public symbols used by the PC XENIX system libraries. In the following program fragments, **_dataitem** and **_func** would be referenced in C without the leading underscore as **dataitem** and **func**.

Main programs in PC XENIX are really functions with the name **_main** (**main** in C) so they follow the same conventions as any other C procedure. Register conventions used by C (already setup) are:

- DS = SS = segment selector value of DGROUP
- ES = some valid selector (might not be same as DS)
- SK = stack pointer

- BP = frame pointer
- SI and DI are assumed to be preserved
- Direction flag is clear (increment mode).

Data segment declarations can have any valid name. A rigid format for data segment names and data segment declarations must be followed when interfacing with C language programs. For more information, see Appendix C, "Segment Names For High-Level Languages."

The following data segment illustrates this format:

```
DGROUP      group       _DATA

_DATA       segment     word public 'DATA'

            assume  ds:DGROUP

            public  _dataitem

_dataitem   dw       0

_DATA       ends
```

The above example associates the group name **DGROUP** with the data segment named **_DATA**. The **assume** directive selects the **ds** register to be the default segment register for all labels and variables defined in the group named **DGROUP**.

Extra data segments (not in DGROUP) may be:

- Any valid name
- Word aligned (for efficiency)
- Class **'FAR_DATA'**. (Preferred—same as C. Any class name not used by a segment in **DGROUP** or a code segment is valid.)

```
DATASEG    segment     word public 'FAR_DATA'

Any far data items (not in DGROUP) are included here

DATASEG    ends
```

Code segments declarations can have any valid name. However, when interfacing with C language programs, a rigid format for code segment names and code segment declarations must be followed. For more information, see Appendix C, "Segment Names For High-Level Languages."

The following program code fragment illustrates this format. This fragment identifies a skeleton of a procedure named **_func**, which is made accessible to other program modules by specifying **_func** in the public directive.

```
extrn     _calledfunc:near ;(replace near with far if
                           ;medium, large, or huge model)

MOD_TEXT  segment   byte public 'CODE'

          assume    cs:MOD_TEXT, ds:DGROUP, ss:DGROUP

public  _func                ;1 underscore

_func   proc    near         ;(replace near with far if
                             ;medium, large, or huge model)
        push    bp
        mov     bp,sp
```

If you have local frame variables, allocate space for them on the stack as follows:

No stack checking:

```
        sub     sp,localframesize
```

With stack checking:

```
        mov     ax,localframesize
        call    __chkstk ; allocate ax bytes for
                         ;localframesize
                         ;and check for stack overflow

        push    si       ; if si is used in _func
        push    di       ; if di is used in _func
```

Insert your code here.

```
        push    12
        push    offset DGROUP:_dataitem
        call    _calledfunc
        add     sp,4        ;need to clear the two words
                            ;pushed on the stack
        pop     di          ; if di was pushed above
        pop     si          ; if si was pushed above
        mov     sp,bp
        pop     bp
        ret                 ;type depends on proc def.
_func     endp

MOD_TEXT  ends
          end              ; (omit a start address
                           ; in the end statement)
```

If **_func** returns a result and is one of the following types, the result is stored according to the type specified below:

- **char**, the value is in **AL**
- **int** (short), the value is in **AX**
- **int** (long), the value is in **DX**, **AX** (**DX** is high order)
- float, double, struc (refer to *XENIX C Library Guide and Compiler Reference*).

In a medium, large, or huge model, declare external code references outside of the code segment. This is because all externals declared inside a segment are assumed to be in that segment.

Refer to *XENIX C Library Guide and Compiler Reference* for more details on how to reference parameters in functions and how to pass parameters to and from C language functions.

Once you have tested the program, you can invoke it from the PC XENIX command line at any time. Programs that you create, like all other PC XENIX programs, can accept command parameters. You also can copy these programs to other systems and invoke them with shell scripts.

Assembling a Source File

You can assemble a program source file by typing the **as** command name and the name of the file you want to process. The command line has this form:

`as [`*options*`]` *filename*

The *options* can be any combination of Macro Assembler options. You can place these options in any order within the options field on the command. We describe the *options* in "Using the Options" on page 1-6.

The *filename* must be the name of the source file you want to assemble.

Unless otherwise specified, the assembler uses a default file name for the relocatable object code. The default file name is the same as the source file, except that **.o** replaces the file name extension.

The assembler also uses a default file name for the listing file (if requested). The assembly listing shows the assembled code for each source statement and the names and types of symbols defined in the program. If you do not request a listing by using the appropriate option, the assembler does not create an assembly listing. The default file name is the same as the source file, except that **.lst** replaces the file name extension.

Using the Options

The Macro Assembler options control the operation of the assembler and the format of the output files it generates. The Macro Assembler has the following options:

-a	Alphabetical ordering for segments
-d	Output a pass 1 listing
-D*sym*	Define symbols
-I*path*	Include file path names
-l[*file*]	Output listing to file or default
-Ml	Preserve case sensitivity in names (default)
-Mu	Map all symbols to uppercase
-Mx	Preserve case sensitivity in externals
-n	Suppress symbol table information
-o[*file*]	Output object code to file or suppress object file
-r	Real floating-point instructions
-v	Output assembly statistics
-x	List errors to standard error channel
-X	List false conditionals.

You can place options in any order within the *options* field of the **as** command.

Alphabetical Ordering for Segments

The **-a** option directs the assembler to place the assembled segments in alphabetical order before copying them to the object file. If you do not use this option, the assembler copies the segments in the order encountered in the source file. For example:

```
as -a file.s
```

This example creates an object file, **file.o**, whose segments are arranged in alphabetic order. Thus, if the source file, **file.s**, contains definitions for the segments **DATA**, **CODE**, and **MEMORY**, the assembled segments in the object file have the order **CODE**, **DATA**, and **MEMORY**.

Output a Pass 1 Listing

The **-d** option directs the assembler to add a pass 1 listing to the assembly listing file, making the assembly listing show the results of both assembler passes. You typically use a pass 1 listing to locate and understand program phase errors. Phase errors occur when the assembler makes assumptions about the program in pass 1 that are not valid in pass 2.

The **-d** option does not create a pass 1 listing unless you also direct the assembler to create an assembly listing. It does direct the assembler to display error messages for both pass 1 and pass 2 of the assembly, even if the assembler does not create an assembly listing. For example:

```
as -d -l file.s
```

This example directs the assembler to create a pass 1 listing for the source file **file.s**. Then the assembler places the listing in the file **file.lst**.

Define Symbols

The **-D** option directs the assembler to define the symbol appended to the **-D** flag as a text macro with a null value. (For a discussion on text macros, see “EQU Directive” on page 4-9.)

The assembler defines the symbol with the case in effect at that point in the command line. Any number of **-D** switches can be used. The defined symbol can be tested with the **IFDEF** and **IFNDEF** directives during the assembly. For example:

```
as -Dsymbol_a file.s
```

This example directs the assembler to define the symbol *symbol_a* as a null **textmacro**. The default, to preserve symbol names in lowercase, will occur in this example.

Include File Path Names

The **-I** option directs the assembler to use the specified path name as a prefix to the file names given in the **INCLUDE** directives in an assembly program. You can specify up to 10 **-I** options on the command line. You can use an **-I** option to force searching of the current directory in a specific order. For example:

```
as -I/usr/include -I. file.s
```

This example forces the **INCLUDE** directive to search **/usr/include**, then the current directory (specified by **-I.**) for the given file name.

Output Listing File

The **-l** option directs the assembler to generate a listing file. If the **-l** flag has a file name appended to it (of the form **-llistfile**), the assembler writes the listing to the file **listfile** rather than to the default listing file whose name is the same as that of the first input file except that it has the extension **.lst**.

The **-l** option directs the assembler to generate a listing file to the standard output file, which is usually the console device. For example:

```
as -l file.s
```

This example directs the assembler to generate a listing in the file **file.lst**.

```
as -llist file.s
```

This example directs the assembler to generate a listing in the file **list**.

Preserve Case Sensitivity in Names

The **-Ml** option directs the assembler to preserve lowercase letters in label, variable, and symbol names when writing them to the object file and assembly listing file. This means that the assembler considers names that have the same spelling but use different case letters unique. For example, with the **-Ml** option, **DATA** and **data** are unique. Under PC XENIX, this is the default case mapping option.

You typically use the **-Ml** option when you are going to link a source file with object modules created by a case-sensitive compiler. For example:

```
as -Ml file.s
```

This example directs the assembler to preserve lowercase letters in any names defined in the source file **file.s**.

Map All Symbols to Uppercase

The **-Mu** option directs the assembler to convert all letters in all symbols to uppercase when writing them to the object file and assembly listing file. For example:

```
as -Mu file.s
```

This example directs the assembler to convert lowercase letters to uppercase in any names defined in the source file **file.s**.

Preserve Case Sensitivity in Externals

The **-Mx** option directs the assembler to preserve lowercase letters in public and external names when writing these names to the object file. For the assembly listing file, the assembler converts all symbols from lowercase letters to uppercase except for public and external names.

Public and external names are any label, variable, or symbol names that have been defined using the **PUBLIC** and **EXTRN** directives. Because the assembler converts all other names to uppercase for assembly, these names must have unique spellings. That is, the names **DATA** and **Data** are not unique.

You use the **-Mx** option to ensure that the names of routines or variables copied to the object module have the correct spelling. Use this option with any source file that you will link with object modules created by a case-sensitive compiler. This option is particularly useful in transporting assembler files from DOS to PC XENIX. For example:

```
as -Mx file.s
```

This example directs the assembler to preserve lowercase letters in any public or external names defined in the source file **file.s**.

Suppress Symbol Table Information

The **-n** option directs the assembler to suppress information about the symbols used in the assembled program. For this option to take effect, you also must use the **-l** option. For example:

```
as -l -n file.s
```

This example directs the assembler to generate a listing file without any symbol information in the file **file.lst**.

The **-o** option directs the assembler to suppress the object file or to generate a specified name file for the object code. The **-o** option without any file name suppresses the generation of an object file. If the **-o** flag has a file name appended to it (of the form **-oobjfile**), the assembler writes the object code to the file **objfile**. If the **-o** option is omitted, the assembler writes the object code to the default file whose name is the same as that of the first input file, except that it has the extension **.o** appended to it. For example:

```
as -o file.s
```

This example directs the assembler to suppress the generation of the object file.

```
as -oObj file.s
```

This example directs the assembler to generate object code in the file **Obj**.

```
as file.s
```

This example directs the assembler to generate object code in the file **file.o**.

Real Floating-Point Instructions

The **-r** option directs the assembler to generate floating-point instruction code that an 8087 or an 80287 coprocessor can execute. Programs you create using the **-r** option can run only on machines having an 8087 or 80287 coprocessor. For example:

```
as -r file.s
```

This example directs the assembler to assemble the source file, **file.s**, and create actual 8087 or 80287 instruction code for floating-point instructions.

If the **-r** option is not entered, the assembler generates the floating-point instruction codes that are changed to software interrupts when you link the program with the standard PC XENIX C libraries. This is the default option under PC XENIX. If an 80287 or 8087 is present, PC XENIX changes the software interrupts into real 80287 or 8087 instructions. If the coprocessor is not present, a software emulator in PC XENIX is used to process the software interrupts as if the 80287 or 8087 actually were present. Unemulated instructions at execution time give a SIGILL signal, which cause the process to be terminated, unless this signal is caught in the user program using the **signal** system call. For example:

```
as file.s
```

This example directs the assembler to create emulation code for any floating-point instructions it finds in the program.

Output Assembly Statistics

The **-v** option directs the assembler to output additional statistical information about the assembled program to the standard output channel. This additional information contains the number of the source lines of the assembly, the number of lines assembled, and the total number of symbols. The statistics are also output. The statistics indicate the number of bytes of symbol space free, the number of warnings, and number of severe errors.

If you select the **-l** option, the assembler also prints the above information. For example:

```
as -v file.s
```

This example directs the assembler to output additional assembly statistics to the standard output channel.

List Errors to Standard Error Channel

The **-x** option directs the assembler to print error messages on the standard error channel without displaying the source line in error. In addition, the assembler prints the messages generated in the listing file. If you specify **-l**, the **-x** option has no effect. By using the **-x** option, the assembler executes faster. Error messages can be completely suppressed by using the **-x** option, which makes assemblies silent; that is, sending no output to the standard error channel. For example:

```
as -x file.s
```

This example directs the assembler to print copies of error messages only, written to the standard error channel.

List False Conditionals

The **-X** option directs the assembler to copy to the assembly listing all statements forming the body of an **IF** directive whose expression (or condition) evaluates to false. If you do not enter the **-X** option in the command line, the assembler suppresses the listing of all such statements. The **-X** option lets you list conditionals that do not generate code. This option applies to all **IF** directives:

IF
IFE
IF1
IF2
IFDEF
IFNDEF
IFB
IFNB
IFIDN
IFDIF

The **.SFCOND**, **.LFCOND**, and **.TFCOND** directives in a source file modify the effect of the **-X** option. The **.SFCOND** directive suppresses false conditionals regardless of whether you enter **-X** in the command line. The **.LFCOND** directive lists false conditionals regardless of whether you enter **-X** in the command line. The **.TFCOND** directive reverses the normal meaning of the **-X** option. When you specify the **-X** option and the assembler encounters a **.TFCOND** directive in a source file, the assembler suppresses all subsequent false conditionals. The next encountered **.TFCOND** directive restores the listing.

The **-X** option does not affect the assembly listing unless you direct the assembler to create an assembly listing file. For example:

```
as -X -l file.s
```

If the source file, **file.s**, does not contain a **.TFCOND** directive, this example directs the assembler to list all false conditionals it finds in the source file.

Reading the Assembly Listing

The assembler creates an assembly listing of your source file whenever you include an assembly listing option (**-l**[*file*]) in the **as** command line. The assembly listing contains a list of the statements in your program and the object code generated for each statement. The listing also prints, in one or more tables, the names and values of all symbols in your source file. These names include macros, structures, records, segments, groups, and other symbols. The assembler places these tables at the end of the assembly listing.

The assembler lists symbols only if it encounters any in the program. If there are no symbols in your program for a particular table, the table is omitted. For example, if you use no macros in your program, you will not see a macro section in the symbol table. Also, no symbol tables are created if the **-n** option is used.

The assembly listing also contains error messages if any errors occur during assembly. The assembler places the messages below the statements that caused the errors. At the end of the listing, the assembler displays the number of error and warning messages it issued.

The following sections explain the format of the assembly listing and the meaning of special symbols used in the listing.

Reading Program Code

The assembler lists the program code generated from the statements of a source file. Each line has this form:

offset code statement

The *offset* is the offset from the beginning of the current segment to the code. The *code* is the actual instruction code or data generated by the assembler for the statement. The assembler gives the hexadecimal value of the code, if possible. Otherwise, it indicates what action needs to be taken to compute the value. The *statement* is the source statement shown exactly as it appears in the source file or after processing by a **MACRO**, **IRP**, or **IRPC** directive.

If any errors occur during assembly, the assembler prints the error message directly below the statement where the error occurred. It displays the source file and line number in addition to the error number and error message.

The assembler uses the special characters in the following list to indicate addresses that need to be resolved by the linker (**ld**) or values that were generated in a special way:

Character	**Meaning**
R	Relocatable address; linker must resolve
E	External address; linker must resolve
---	Segment/group address; linker must resolve
=	EQU or = directive
nn:	Segment override in statement
nn/	REP or LOCK prefix instruction
nn [*xx*]	DUP expression; *nn* copies of the value *xx*
+	Macro expansion
C	Included line from INCLUDE file

For example:

```
Macro Assembler                         January 17 08:57:28 1986
                                        Page     1-1


0000                        _DATA segment word public 'DATA'
0000 48 65 6C 6C 6F 2E 0A   hello  db    "Hello.",10
0007 2F 64 65 76 2F 74 74   tty    db    "/dev/tty",0
     79 00
0010 0000                   fd     dw    0
0012                        _DATA  ends

                            DGROUP  group _DATA

                            extrn _open:near
                            extrn _close:near
                            extrn _write:near
                            extrn _exit:near


0000                        _TEXT segment byte public 'CODE'
                                  assume cs:_TEXT,ds:DGROUP
                                  assume ss:DGROUP,es:DGROUP
                                  public _main

0000                        _main        ;program entry point

0000 6A 02                        push 2  ;fd=open("/dev/tty",21)
0002 68 0007 R                    push offset DGROUP:tty
0005 E8 0000 E                    call _open
```

```
0008 83 C4 04                     add sp,4
000B A3 0010 R                    mov fd,ax

000E 6A 07                        push 7  ;write(fd,&hello,7)
0010 68 0000 R                    push offset DGROUP:hello
0013 FF 36 0010 R                 push fd
0017 E8 0000 E                    call _write
001A 83 C4 06                     add sp,6

001D FF 36 0010 R                 push fd ;close(fd)
0021 E8 0000 E                    call _close
0024 83 C4 02                     add sp,2

0027 6A 00                        push 0  ;exit(0)
0029 E8 0000 E                    call _exit

002C                        _TEXT ends

                                  end
```

Reading a Macro Table

The assembler lists the names and sizes of all macros defined in a source file. The list has two columns: Name and Length.

The Name column lists the names of all macros. The names appear in alphabetic order, spelled exactly as given in the source file. The assembler truncates names that are longer than 31 characters.

The Length column lists the size of the macro in terms of the number of nonblank lines. This size is in hexadecimal numbers. For example:

```
   Name         Length
BIOSCALL        0002
DISPLAY         001C
DOSCALL         0002
KEYBOARD        0003
LOCATE          001A
SCROLL          0004
```

Reading a Structure and Record Table

The assembler lists the names and dimensions of all structures and records in a source file. The table contains two sets of overlapping columns. The Width and # Fields columns list information about the structure or record. The Shift, Width, Mask, and Initial columns list information about the structure or record members.

The Name column lists the names of all structures and records. The assembler lists the names in alphabetic order and spells them exactly as they are given in the source file. The assembler truncates names longer than 31 characters.

For a structure, the Width column lists the size (in bytes) of the structure. The # Fields column lists the number of fields in the structure. Both values are in hexadecimal numbers.

For fields of structures, the Shift column lists the offset (in bytes) from the beginning of the structure to the field. This value is in hexadecimal numbers. The other columns are not used.

```
     Name      Width    # Fields
               Shift    Width  Mask   Initial


PARMLIST       001C      0004
 BUFSIZE       0000
 NAMESIZE      0001
 NAMETEXT      0002
 TERMINATOR    001B
```

For a record, the Width column lists the size (in bytes) of the record. The # Fields column lists the number of fields in the record.

For fields in a record, the Shift count lists the offset (in bits) from the lower order bit of the record to the first bit in the field. The Width column lists the number of bits in the field. The Mask column lists the maximum value of the field, expressed in hexadecimal numbers. The Initial column lists the initial value of the field, if any. For each field, the table shows the mask and initial values as if they were placed in the record and all other fields were set to 0. For example:

```
        Name            Width    # Fields
                        Shift    Width    Mask      Initial


REC0                    0008      0003
        FLD1            0006      0002     00C0      0040
        FLD2            0003      0003     0038      0000
        FLD3            0000      0003     0007      0003
REC1                    000B      0002
        FL1             0003      0008     07F8      0400
        FL2             0000      0003     0007      0002
```

Reading a Segment and Group Table

The assembler lists the names, sizes, and attributes of all segments and groups in a source file. The list has five columns: Name, Size, Align, Combine, and Class.

The Name column lists the names of all segments and groups. The names in the list are given in alphabetic order, except that the names of segments belonging to a group are placed under the group name. Names are spelled exactly as given in the source file. The assembler truncates names longer than 31 characters.

The Size column lists the size (in bytes) of each segment. Because a group has no size, only the word **GROUP** is shown. The size, if given, is in hexadecimal numbers.

The Align column lists the alignment type of the segment. The types can be any of the following:

BYTE
WORD
PARA
PAGE

If the segment is defined with no explicit alignment type, the assembler lists the default alignment for that segment.

The Combine column lists the combine type of the segment. The types can be any one of the following:

NONE
PUBLIC
STACK
MEMORY
COMBINE

NONE is given if no explicit combine type is defined for the segment. **NONE** represents the private combine type.

The Class column lists the class name of the segment. The name is spelled exactly as given in the source file. If no name is given, none is shown. For example:

```
   Name                 Size      Align     Combine     Class

AAAXQQ                  0000       WORD      NONE       'CODE'
DGROUP                  GROUP
 DATA                   0024       WORD      PUBLIC     'DATA'
 STACK                  0014       WORD      STACK      'STACK'
 CONST                  0000       WORD      PUBLIC     'CONST'
 HEAP                   0000       WORD      PUBLIC     'MEMORY'
 MEMORY                 0000       WORD      PUBLIC     'MEMORY'
EXTXCM                  0037       WORD      NONE       'CODE'
MAIN_STARTUP            007e       PARA      NONE       'MEMORY'
```

Reading a Symbol Table

The assembler lists the names, types, values, and attributes of all symbols in the source file. The table has four columns: Name, Type, Value, and Attr.

The Name column lists the names of all symbols. The assembler lists the names in alphabetic order and spells them exactly as given in the source file. The assembler truncates names longer than 31 characters.

The Type column lists the type for each symbol. A type is given as one of the following:

L NEAR	A near label
L FAR	A far label
N PROC	A near procedure label
F PROC	A far procedure label
Number	An absolute number
Alias	An alias for another symbol
Opcode	An instruction opcode
Text	A memory operand, string, or other value.

If Type is **Number**, **Opcode**, **Alias**, or **Text**, the symbol is defined by an **EQU** directive or an = directive. The Type column also lists the length of the symbols that are known. A length is given as one of the following:

BYTE	One byte (8 bits)
WORD	One word (16 bits)
DWORD	Doubleword (2 words)
QWORD	Quadword (4 words)
TBYTE	Ten-bytes (5 words).

A length also can be given as a number. In this case, the symbol is a structure, and the number defines the length (in bytes) of the structure. For example, the type:

```
L 0031
```

identifies a label to a structure that is 31 hexadecimal bytes long.

The Value column shows the numeric value of the symbol. For absolute symbols, the value represents an absolute number. For labels and variable names, the value represents that item's offset from the beginning of the segment in which it is defined. If **Type** is **Number**, **Opcode**, **Alias**, or **Text**, the **Value** column shows the symbol's value, even if the value is simple text.

- **Number** shows a constant numeric value.
- **Opcode** shows a blank (the symbol is an alias for an instruction mnemonic).
- **Alias** shows the name of another symbol.
- **Text** shows the text the symbol represents. Text is any operand that does not fit one of the other three categories.

The Attr column lists the attributes of the symbol. The attributes include the name of the segment in which the symbol is defined, if any, the scope of the symbol, and the code length. The scope of a symbol is given only if the symbol is defined using the **EXTRN** or **PUBLIC** directives. The scope can be **External** or **Global**. The code length is given only for procedures.

For example:

```
Symbols:

Name            Type    Value   Attr

SYM0             Number   0005
SYM1            Text     1.234
SYM2            Number   0008
SYM3            Alias    SYM4
SYM4            Text     5[bp][di]
SYM5            Opcode
SYM6            L BYTE   0002   DATA
SYM7            L WORD   0012       DATA    Global
SYM8            L DWORD  0022       DATA
SYM9            L QWORD  0000       External
LAB0            L FAR    0000       External
LAB1            L NEAR   0010   CODE
```

Reading a Pass 1 Listing

When you specify the **-d** and **-l** options in the **as** command line, the assembler adds a pass 1 listing to the assembly listing file, making the listing file show the results of both assembler passes. The listing is intended to help locate the source of phase errors.

The following examples illustrate the pass 1 listing for a source file named **xx.s** that assembled without error. Although the code produces an error on pass 1, the assembler corrects the error on pass 2 and completes the assembly correctly.

During pass 1, a **JLE** instruction to a forward reference produces an error message:

```
0017  7E 00                JLE     SMLSTK
xx.s(0017) : error 9: Symbol not defined SMLSTK
0019  BB 1000              MOV     BX,4096
001C             SMLSTK:
```

The assembler displays this error because it has not yet encountered the definition for the symbol **SMLSTK**.

By pass 2, **SMLSTK** has been defined, and the assembler can fix the instruction so no error occurs:

```
0017  7E 03                    JLE     SMLSTK
0019  BB 1000                  MOV     BX,4096
001C             SMLSTK:
```

The **JLE** instruction's code now contains **03** instead of **00**. This is a jump of 3 bytes.

Because the assembler generated the same amount of code for both passes, there was no phase error. If a phase error had occurred, the assembler would have displayed an error message.

In the following program fragment, a mistyped label creates a phase error. In pass 1, the label **go** is used in a forward reference and creates a **Symbol not defined** error. The assembler assumes that the symbol will be defined later and generates 3 bytes of code, reserving 2 bytes for the symbol's actual value.

```
0000               _TEXT segment public byte public 'CODE'
0000 E9 0000 U           jmp go
xx.s(0000) : error 9: Symbol not defined go
0003                     go      label   byte
0003 B8 0001             mov     ax, 1
0006               _TEXT ends
```

In pass 2, the label **go** is known to be a label of **BYTE** type, which is an illegal type for the **JMP** instruction. As a result, the assembler produces only two bytes of code in pass 2, one less than in pass 1. The result is a phase error:

```
0000               _TEXT segment public byte public 'CODE'
0003 R                   jmp go
xx.s(0003) : error 57: Illegal size for item
0003                     go      label   byte
xx.s(0003) : error 6: Phase error between passes
0003 B8 0001             mov     ax, 1
0006               _TEXT ends
```

Chapter 2. Elements of the Assembler

Characters, Numbers, and Names

All assembler language programs consist of a combination of characters, numbers, and names:

Names and ***numbers***	Identify values in instruction statements.
Characters	Form the names or number, or form the character constants.

The following sections describe the characters you can use in a program and how you can form numbers and names using these characters.

Character Set

The assembler recognizes the following character set:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
? @ _ $ : . [ ] ( ) < > { } + - / * & % ! ' ~ ¦ \ =
# ^ ; , ` "
```

Integers

Syntax

digits
*digits***B**
*digits***Q**
*digits***O**
*digits***D**
*digits***H**

Description

An integer represents an integer number. It is a combination of binary, octal, decimal, or hexadecimal *digits* and an optional radix. The *digits* are a combination of one or more digits of the specified radix: B, Q, O, D, or H. If you do not give a radix, the assembler uses the current default radix (typically decimal). The following table lists the digits that you can use with each radix. (You can specify radix in either uppercase or lowercase.)

Radix	**Type**	**Digits**
B	Binary	0 1
Q	Octal	0 1 2 3 4 5 6 7
O	Octal	0 1 2 3 4 5 6 7
D	Decimal	0 1 2 3 4 5 6 7 8 9
H	Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

Hexadecimal numbers must always start with a decimal digit (0-9). You can give the hexadecimal digits A through F in either uppercase or lowercase. For example, you must specify the hexadecimal number **AF** by the characters `0af`, `0AF`, `0Af`, or `0aF`.

The maximum number of digits in an integer depends on the instruction or directive in which you use the integer.

You can set the default radix by using the **.RADIX** directive. See ".RADIX Directive" on page 9-2 for more information. For example:

Binary	Octal	Hexadecimal	Decimal	Decimal (by default)
01011010B	132Q	5AH	90D	90
01111B	17O	0FH	15D	15

Real Numbers

Syntax

digits.digits.[**E** [+ ¦-] *digits*]

Description

A real number represents a number having an integer, a fraction, and an exponent. The *digits* can be any combination of decimal digits. Digits before the decimal point (**.**) represent the integer part and those after the point (**.**) represent the fraction. The digits after the exponent mark (**E**) represent the exponent. The exponent is optional. If you give an exponent, you can use the optional plus (+) and minus (-) signs to indicate its value.

You can use real numbers only with the **DD**, **DQ**, and **DT** directives. The maximum number of digits in the number and the maximum range of exponent values depends on the directive. For example:

```
25.23   2.523E1   2523.0E-2
```

Encoded Real Numbers

Syntax

*digits*R

Description

An encoded real number is an 8-, 16-, or 20-digit hexadecimal number that represents a real number in encoded format. An encoded real number has a sign field, a biased exponent, and a mantissa. The assembler encodes these values as bit fields within the number. The exact size and meaning of each bit field depends on the number of bits in the number. The *digits* must be hexadecimal digits. The number must begin with a decimal digit (0-9).

You can use encoded real numbers only with the **DD**, **DQ**, and **DT** directives. The maximum number of digits for the encoded numbers you use with **DD**, **DQ**, and **DT** must not exceed 8, 16, and 20 digits, respectively. (If you include a leading 0, the number must not exceed 9, 17, and 21 digits, respectively.) For example:

```
3F800000R                       ; 1.0 for DD
3FF0000000000000R               ; 1.0 for DQ
3FFF80000000000000000R          ; 1.0 for DT
```

Packed Decimal Numbers

Syntax

`[+|-]` *digits*

Description

A packed decimal number represents a decimal integer that the assembler stores in packed decimal format. Packed decimal storage has a leading sign byte and 9 value bytes. Each value byte contains 2 decimal digits. The high-order bit of the sign byte is 0 for positive values, and 1 for negative values.

Packed decimals have the same format as other decimal integers, except that they can take an optional plus (+) or minus (-) sign. You can use packed decimals only with the **DT** directive. A packed decimal must not have more than 18 digits. For example:

```
 1234567890                 ; encoded as 00000000001234567890
-1234567890                 ; encoded as 80000000001234567890
```

Character and String Constants

Syntax

'*characters*'
"*characters*"

Description

A character constant is a constant composed of a single ASCII character. A string constant is a constant composed of two or more ASCII characters. You must enclose the constant in matching single quotation marks or double quotation marks.

You must encode single quotation marks twice when you use them in constants that you enclose by single quotation marks.

Similarly, you must encode double quotation marks twice when you use them in constants that you enclose by double quotation marks. For example:

```
'a'
'ab'
"a"
"This is a message."
'Can''t find the file.'
"Specified ""value not found."
```

Names

Syntax

characters . . .

Description

A name is a combination of letters, digits, and special characters you can use in instruction statements to indicate labels, variables, and symbols. Names have the following formatting rules:

- A name must begin with a letter, an underscore (_), a question mark (?), a dollar sign ($), or an at sign (@).
- A name can have any combination of uppercase and lowercase letters. To specify case conversions of names that are contained in the assembly listing and the generated object file, refer to the **-M** options in "Using the Options" on page 1-6.
- A name can have any number of characters, but the assembler only uses the first 31 characters. It ignores all other characters.

For example:

```
subrout3
Array
_main
```

Reserved Names

A ***reserved name*** is any name that has a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and predefined group and segment names. You can use these names only as the assembler defines them. You must not redefine these names.

The following is a list of all reserved names, except instruction mnemonics. For a complete list of instruction mnemonics, see Appendix A, "Instruction Summary."

%OUT
.186
.286c
.286p
.287
.8086
.8087
=
AH
AL
AND
ASSUME
AX
BH
BL
BP
BX
BYTE
CH
CL
COMMENT
.CREF
CS
CX
DB
DD
DH
DI
DL
DQ
DS
DT
DW
WORD
DX
ELSE
END
ENDIF
ENDM
ENDP
ENDS
EQ
EQU
ES
EVEN
EXITM
EXTRN
FAR
GE
GROUP
GT
HIGH
IF
IF1
IF2
IFB
IFDEF
IFDIF
IFE
IFIDN
IFNB
IFNDEF
INCLUDE
IRP
IRPC
LABEL
.LALL
LE
LENGTH
.LFCOND
.LIST
LOCAL
LOW
LT
MACRO
MASK
MOD
NAME
NE
NEAR
NOT
OFFSET
OR
ORG
PAGE
PROC
PTR
PUBLIC
PURGE
QWORD
.RADIX
RECORD
REPT
.SALL
SEG
SEGMENT
.SFCOND
SHL
SHORT
SHR
SI
SIZE
SP
SS
STRUC
SUBTTL
TBYTE
.TFCOND
THIS
TITLE
TYPE
.TYPE
WIDTH
WORD
.XALL
.XCREF
.XLIST
XOR

The assembler considers all uppercase and lowercase combinations of these names to be the same name. For example, the names Length and LENGTH are the same name for the LENGTH operator.

Statements and Comments

All assembler language source files consist of one or more statements. ***Statements*** define the actions the assembler takes, such as generating instruction code or declaring of a variable.

Assembler language source files also can contain comments. ***Comments*** are text you supply that describes the action of the program or the purpose of declared variables or labels.

The following sections describe the format of statements and comments in detail.

Statements

Syntax

[*name*] *mnemonic* [*operands*]

Description

A statement is a combination of a name, an instruction or directive mnemonic, and one or more operands. A statement represents an action for the assembler to take, such as generating a machine instruction, generating one or more bytes of data, or copying a program listing to a file.

Statements have the following formatting rules:

- A statement can begin in any column.
- A statement must not be more than 128 characters in length and must not contain an embedded newline character. This means that you cannot continue a statement on multiple lines.
- You must end a statement by a newline character. This includes the last statement in the source file.

For example:

```
count   db      0
mov     ax, bx
assume cs:_TEXT, ds:DGROUP
_main   proc    far
```

Comments

Syntax

;text

Description

A comment is any combination of characters you precede by a semicolon (;) and terminate with a newline character. Comments let you describe the action of a program at the given point. The assembler otherwise ignores comments. The comments have no effect on assembly.

You can place comments anywhere in a program, including on the same line as a statement. You must place the comment after you give all names, mnemonics, and operands. A comment must not be longer than one line; that is, it must not contain any embedded new-line characters. For very long comments, you can use the **COMMENT** directive. For example:

```
; This comment is alone on a line.
mov    ax, bx  ;This comment follows a statement.
; Comments can contain reserved words like PUBLIC.
```

COMMENT Directive

Syntax

COMMENT *delim text delim*

Description

The **COMMENT** directive causes the assembler to treat all *text* between the given pair of delimiters (*delim*) as a comment. The delimiter character must be the first nonblank character after the **COMMENT** keyword. The *text* is all remaining characters up to the next occurrence of the delimiter. The *text* must not contain the delimiter.

You typically use the **COMMENT** directive for multiple-line comments. Although text can appear on the same line as the last delimiter, the assembler ignores any text after the delimiter. For example:

```
comment *
This comment continues until the
next asterisk.
*

comment +
The assembler ignores the
following MOV statement
+ mov   ax, 1
```

Chapter 3. Program Structure

Introduction

The program structure directives let you define the organization of a program's code and data when they load into memory.

Here are the program structure directives:

SEGMENT	Segment Definition
ENDS	Segment End
END	Source File End
GROUP	Segment Groups
ASSUME	Segment Registers
ORG	Segment Origin
EVEN	Segment Alignment
PROC	Procedure Definition
ENDP	Procedure End

The following sections describe these directives in detail. They also describe the instruction set directives that define which instruction set the assembler uses during assembly.

Source Files

Every assembly language program consists of one or more source files. A source file is a text file that contains statements that define the program's data and instructions. The assembler reads source files and assembles the statements to create object modules that the system linker can prepare for execution.

All source files have the same form—zero or more program segments followed by an END statement. Every source file requires the **END** statement to signal the end of the source file. This statement also provides a way to define the program entry point or starting address, but this option is not used for the PC XENIX system assemblies. You can use it for creating code for use by DOS. All other statements in a source file are optional.

The following example illustrates the source file format. It is a complete assembly language program that uses PC XENIX system calls to print the message `Hello.` on the system display. Linking the module with the standard C runtime library produces a complete executable program.

```
_DATA    segment  word public 'DATA' ;program data
hello   db      "Hello.", 10
tty     db      "/dev/tty", 0
fd      dw      0
_DATA    ends

DGROUP group  _DATA

extrn  _open:near                    ;external entry points
extrn  _close:near
extrn  _write:near
extrn  _exit:near

_TEXT   segment byte public 'CODE'   ;program code segment
        assume cs:_TEXT, ds:DGROUP, ss:DGROUP, es:DGROUP
        public _main

_main:                               ;program entry point

        push    2                    ;fd = open("/dev/tty", 2)
        push    offset DGROUP:tty
        call    _open
        add     sp, 4
        mov     fd, ax

        push    7                    ;write(fd, &hello, 7)
        push    offset DGROUP:hello
        push    fd
        call    _write
        add     sp, 6

        push    fd                   ;close(fd)
        call    _close
        add     sp, 2

        push    0                    ;exit(0)
        call    _exit

_TEXT   ends

        end
```

The main features of this source file are:

- The **SEGMENT** and **ENDS** statements, defining segments named **_DATA** and **_TEXT**.
- The **GROUP** statement defining a group **DGROUP**, which contains the data segment, **_DATA**.
- The variables `hello` and `tty` in the **_DATA** segment, defining the string to be displayed and the name of the file that is opened to do this.
- The instruction label **_main** in the **_TEXT** segment and its **PUBLIC** declaration. These provide the necessary entry point for the runtime library to call.
- The **assume** statements in the **_DATA** and **_TEXT** segments, defining which segment registers will be associated with the labels, variables, and symbols defined within the segments.

Instruction Set Directives

Syntax

```
.8086
.8087
.186
.286c
.286p
.287
```

Description

The instruction set directives enable and disable the instruction sets for the given microprocessors. When you specify a directive, the assembler recognizes and assembles any subsequent instructions belonging to that microprocessor. If you use the instruction set directives, place them at the beginning of the program source file. This ensures that the assembler assembles all instructions in the file using the same set.

Under PC PC XENIX, the assembler assembles nonprotected 80286 and 80287 instructions by default, so the **.286c** and **.287** directives are not required.

The **.8086** directive enables assembly of instructions for the 8086 microprocessor. It also disables assembly of 80186 and 80286 instructions.

The **.8087** directive enables assembly of instructions for the 8087 floating-point coprocessor and disables assembly of 80287 instructions.

The **.186** directive enables assembly of instructions for the 80186 microprocessor. Use this directive for programs that a 80186 microprocessor will execute.

The **.286c** directive enables assembly of nonprotected instructions for the 80286 microprocessor. (These are identical to the 80186 instructions.)

The **.286p** directive enables assembly of the protected instructions of the 80286.

The **.287** directive enables assembly of instructions for the 80287 floating-point coprocessor. Use this directive with programs that have floating-point instructions and that execute on an 80286 microprocessor. It is not necessary to include the **.287** directive because the assembler assembles nonprotected 80287 instructions by default.

Even though a source file may contain the **.8087** or **.287** directive, the assembler also requires the option in the **as** command line to define how to assemble floating-point instructions. The **-r** option directs the assembler to generate the actual instruction code for the floating-point instruction. Without the **-r** option, the assembler directs it to generate the instruction codes, which are changed into actual floating-point instruction codes as software interrupts at program link time.

SEGMENT and ENDS Directives

Syntax

name SEGMENT *align combine* '*class*'
name ENDS

Description

The **SEGMENT** and **ENDS** directives mark the beginning and end of a program segment. A program segment is a collection of instructions, data, or both, whose addresses are all relative to the same segment register.

The *name* defines the name of the segment. This name can be unique or be the same name given to other segments in the program. The assembler treats segments with identical names as the same segment.

The *align*, *combine*, and *class* options define program loading instructions that the linker uses when forming the executable program. These options are described later.

You can nest segments. When the assembler encounters a nested segment, it temporarily suspends assembly of the enclosing segment, and begins assembly of the nested segment. When it finishes assembly of the nested segment, the assembler continues assembly of the enclosing segment. You cannot use overlapping segments. For example:

```
SAMPLE_TEXT     segment byte public 'CODE'
_main proc far
.
.
.
CONST segment word public 'CONST'      ;nested segment
seg1  dw      ARRAY_DATA
CONST ends                             ;end nesting

      mov     es, seg1
      push    es
      mov     ax,es:pointer
      push    ax
      call    _printf
      add     sp,4
      .
      .
      .
      ret
_main endp
SAMPLE_TEXT     ends
      end
```

This example contains two segments: **SAMPLE_TEXT** and **CONST**. The **CONST** segment is nested within the **SAMPLE_TEXT** segment.

You can use a given segment name more than once in a source file. Each segment definition using that name, however, must have either exactly the same attributes or attributes that do not conflict.

Program Loading Options

The *align* option defines the alignment of the given segment. The alignment defines the range of memory addresses from which you can select a starting address for the segment. It can be any one of the following:

BYTE	Use any byte address.
WORD	Use any word address (2 bytes/word).
PARA	Use paragraph addresses (16 bytes/paragraph).
PAGE	Use page addresses (1024 bytes/page).

If you do not give an *align*, the assembler uses **PARA** by default. The actual start address is computed when the program loads, and the linker guarantees that the address will be on the given boundary.

The optional *combine* defines how to combine segments having the same name. It can be any one of the following:

PUBLIC	Concatenates all segments having the same name and forms a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and the assembler adjusts all offsets to represent the distance from the beginning of the new segment.
STACK	Concatenates all segments having the same name and forms a single, contiguous segment. All addresses in the new segment are relative to the SS segment register. The Stack Pointer (SP) register is set to an address in the segment.
COMMON	Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address.
MEMORY	Places all segments having the same name in the highest physical segment in memory. If you give more than one **MEMORY** segment, the assembler overlaps the segments, as with **COMMON** segments.
AT *address*	Causes all label and variable addresses defined in the segment to be relative to the given *address*. The *address* can be any valid expression but must not contain a forward reference, that is, a reference to a symbol defined later in the source file. AT segments typically contain no code or initialized data.

Instead, they represent address templates that can be placed over code or data already in memory. You then can use the labels and variables in the AT segments to access the fixed instructions and data.

If you do not specify a *combine* attribute, the segment is not combined. Instead, it receives its own physical segment when it loads into memory.

Note: The linker requires at least one stack segment in a program. The stack segment is supplied in the standard PC XENIX library (that is, **/lib/Sseg.o** for a small module program).

The *class* option defines which segments are to be loaded in contiguous memory. Segments having the same class name load into memory one after another. All segments of a given class load before segments of any other class. You must enclose the *class* name in single quotation marks. It must be in uppercase. For example:

```
      assume  cs:_TEXT
_TEXT segment word public 'CODE'
.
.
.
_TEXT ends
```

This example illustrates the general form of a text segment for a small module program. The segment name is **_TEXT**. The segment alignment and combine type are **word** and **public**, respectively. The class is **CODE**.

Note: For information on high-level languages interfaces, refer to Appendix C, "Segment Names For High-Level Languages."

END Directive

Syntax

END *expression*

Description

The **END** directive marks the end of the module. The assembler ignores any statements following this directive.

For assemblies, the expression option should be blank. A label called **_main** must be included in the main program and is recognized by the assembler and the linker as the entry point.

The *expression* option defines the program entry point for programs that are coded in PC PC XENIX and are to be assembled under DOS. The entry point defines the address at which program execution is to start. If the program has more than one module, only one of these modules can define an entry point. The module with the entry point is the ***main module.*** If you do not give an entry point, the assembler does not assume one. For example:

```
end
end      _main
```

GROUP Directive

Syntax

name GROUP *seg-name*,,,

Description

The **GROUP** directive associates a group *name* with one or more segments. It causes all labels and variables defined in the given segments to have addresses. These addresses are relative to the beginning of the group instead of to the beginning of the segments in which they are defined. The *seg-name* must be the name of a segment defined using the **SEGMENT** directive or a **SEG** expression. The *name* must be unique.

The **GROUP** directive does not affect the order in which the assembler loads segments of a group. The loading order depends on each segment's class or on the order in which the linker receives the object modules.

Segments in a group do not have to be contiguous. This means that segments that do not belong to the group can be loaded between segments that do. There is one restriction. The distance between the first byte in the first segment and the last byte in the last segment must not exceed 65,535 bytes. If the segments of a group are contiguous, the group can occupy up to 64K of memory.

You can use group names with the **ASSUME** directive and as an operand prefix with the segment override operator (:).

Note: Do not use a group name in more than one **GROUP** directive in any source file. If several segments within the source file belong to the same group, give all segment names in the same **GROUP** directive. For example:

```
DGROUP       group   _DATA, _BSS
      assume ds:DGROUP

_DATA segment word public 'DATA'
.
.
.
_DATA ends
_BSS  segment word public 'BSS'
.
.
.
_BSS  ends
      end
```

ASSUME Directive

Syntax

ASSUME *seg-reg*:*seg-name*,,,
ASSUME NOTHING

Description

The **ASSUME** directive selects the given segment register *seg-reg* to be the default segment register for all labels and variables defined in the segment or group given by *seg-name*. Subsequent references to the label or variable automatically assume the selected register when the assembler computes the effective address.

The ASSUME directive can define up to four selections: one selection for each of the four segment registers. The *seg-reg* can be any one of the segment register names: CS, DS, ES, or SS. The *seg-name* must be one of the following:

- The name of a segment previously defined with the **SEGMENT** directive.
- The name of a group previously defined with the **GROUP** directive.
- The keyword **NOTHING**.

The keyword **NOTHING** cancels the current segment selection. The directive **ASSUME NOTHING** cancels all register selections made by a previous **ASSUME** statement.

Note: You can use the segment override operator (:) to override the current segment register the **ASSUME** directive selects. For example:

```
assume cs:code
assume cs:cgroup,ds:dgroup,ss:nothing,es:nothing
assume nothing
```

ORG Directive

Syntax

`ORG` *expression*

Description

The **ORG** directive sets the location counter to *expression.* Subsequent instruction and data addresses begin at the new value.

The *expression* must resolve to an absolute number; that is, the assembler must know all symbols in the expression on pass 1. You also can use the location counter symbol ($). For example:

```
org     120H
org     $+2
```

EVEN Directive

Syntax

`EVEN`

Description

The **EVEN** directive aligns the next data or instruction byte on a word boundary. If the current value of the location counter is odd, the directive increments the location counter to an even value and generates one NOP instruction (90H). If the location counter is already even, the assembler ignores the directive.

Do not use the **EVEN** directive in byte-aligned segments. For example:

```
        org     0
test1   db      1
        even
test2   dw      513
```

In this example, the **EVEN** directive increments the location counter and generates a **NOP** instruction (90H). This means the offset of `test2` is 2, not 1.

PROC and ENDP Directives

Syntax

name PROC *type*
      statements
name ENDP

Description

The **PROC** and **ENDP** directives mark the beginning and end of a procedure. A procedure is a block of instructions that form a program subroutine.

Every procedure has a *name* with which it can be called. The *name* must be a unique name, not previously defined in the program. The optional *type* can be either **NEAR** or **FAR**. The assembler assumes **NEAR** if you do not specify a *type*. The *name* has the same attributes as a label, and you can use it as an operand in a **jump**, **call**, or **loop** instruction.

Any number of *statements* can appear between the **PROC** and **ENDP** statements. The procedure should contain at least one **ret** statement to return control to the point of call. The assembler allows nested procedures. For example:

```
_main   proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
        mov     ax,offset DGROUP:string
        push    ax
        call    _printf
        add     sp,2
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret
_main   endp
```

Chapter 4. Types and Declarations

Label Declarations

Label declarations create labels. A ***label*** is a name that represents the address of a given instruction. You can use labels in **JMP**, **CALL**, and other execution control instructions to direct program execution to the associated instruction.

Near Label Declarations

Syntax

name:

Description

A near label declaration creates an instruction label that has NEAR type. You can use the label in subsequent instructions in the same segment to pass execution control to the corresponding instruction.

The *name* must be unique and not previously defined. Furthermore, the segment containing the declaration must be associated with the **CS** segment register (see the **ASSUME** directive). The assembler sets the name to the current value of the location counter.

A near label declaration can appear on a line by itself or on a line with an instruction as long as it immediately precedes the instruction. For example:

```
start:
loop: inc     4[bp]
```

Procedure Labels

Syntax

name `PROC [ NEAR | FAR ]`

Description

The **PROC** directive creates a label *name* and sets its type to **NEAR** or **FAR**. The label then represents the address of the following instruction. You can use the label in **JMP**, **CALL**, or **LOOP** instruction to direct execution control to the given instruction.

When the assembler encounters the **PROC** label definition, the assembler sets the label's value to the value of the current location counter and sets its type to **NEAR** or **FAR**. If the label has **FAR** type, the assembler also sets its segment value to that of the enclosing segment.

You can use **NEAR** labels with **JMP**, **CALL**, and **LOOP** instruction in the enclosing segment only. You can use **FAR** labels in any segment of the program.

Data Declarations

The ***data declaration directives*** let you generate data for a program. The directives translate numbers, strings, and expressions into individual bytes, words, or other units of data. The encoded data is copied to the program object file.

Here are the data declaration directives:

DB	Data byte
DW	Data word
DD	Data doubleword
DQ	Data quadword
DT	Data ten-byte word

The following sections describe these directives in detail.

DB Directive

Syntax

[*name*] DB *initial-value*,,,

Description

The DB directive allocates and initializes a byte (8 bits) of storage for each given *initial-value*. The *initial-value* can be:

- An integer
- A character string constant
- A **DUP** operator
- A constant expression
- A question mark (?).

The question mark (?) represents an undefined initial value. If you give two or more initial values, separate them by commas (,).

The *name* is optional. If you give a *name*, the directive creates a variable of type **BYTE** whose offset value is the current location counter value.

A string constant can have any number of characters as long as it fits on a single line. When the string is encoded, the characters are stored in the order given, with the first character in the constant at the lowest address and the last at the highest. For example:

```
integer         db      16
string          db      'ab'
message         db      "Enter your name: "
constantexp     db      4 * 3
empty           db      ?
multiple        db      1, 2, 3, '$'
duplicate       db      10 dup(?)
high_byte       db      255
```

DW Directive

Syntax

[*name*] DW *initial-value*,,,

Description

The **DW** directive allocates and initializes a word (2 bytes) of storage for each given *initial-value*. An *initial-value* can be:

- An integer
- A string constant
- A **DUP** operator
- A constant expression
- An address expression
- A question mark (?).

The question mark (?) represents an undefined initial value. If you give two or more expressions, separate them by commas (,).

The *name* is optional. If you give a *name*, the directive creates a variable of type **WORD** whose offset value is the current location counter value.

String constants must not exceed 2 characters in length. The assembler places the last (or only) character in the string in the low-order byte. It also places either zero or the first character in the high-order byte. For example:

```
integer         dw      16728
character       dw      'a'
string          dw      'bc'
constantexp     dw      4 * 3
addressexp      dw      string
empty           dw      ?
multiple        dw      1, 2, 3, '$'
duplicate       dw      10 dup(?)
high_word       dw      65535
arrayptr        dw      array
arrayptr2       dw      offset DGROUP:array
```

DD Directive

Syntax

[*name*] DD *initial-value*,,,

Description

The **DD** directive allocates and initializes a doubleword (4 bytes) of storage for each given *initial-value.* An *initial-value* can be:

- An integer
- A real number
- A one- or two-character string constant
- An encoded real number
- A **DUP** operation
- A constant expression
- An address expression
- A question mark (?).

The question mark (?) represents an undefined initial value. If you give two or more expressions, you must separate them by commas (,).

The *name* is optional. If you give a *name*, the directive creates a variable of type **DWORD** whose offset value is the current location counter value.

String constants must not exceed 2 characters in length. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeros are placed in all remaining bytes. For example:

```
integer         dd      16728
character       dd      'a'
string          dd      'bc'
real            dd      1.5
encodedreal     dd      3f000000R
constantexp     dd      4 * 3
addsegexp       dd      real
empty           dd      ?
multiple        dd      1, 2, 3, '$'
duplicate       dd      10 dup(?)
high_double     dd      4294967295
```

DQ Directive

Syntax

[*name*] DQ *initial-value*,,,

Description

The **DQ** directive allocates and initializes a quadword (8 bytes) of storage for each given *initial-value*. An *initial-value* can be:

- An integer
- A real number
- A one- or two-character string constant
- An encoded real number
- A **DUP** operator
- A constant expression
- A question mark (?).

The question mark (?) represents an undefined initial value. If you give two or more expressions, separate them by commas (,).

The *name* is optional. If you give a *name*, the directive creates a variable of type **QWORD** whose offset value is the current location counter value.

String constants must not exceed 2 characters in length. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are 2 in the string) is placed in the next byte. Zeros are placed in all remaining bytes. For example:

```
integer         dq      16728
character       dq      'a'
string          dq      'bc'
real            dq      1.5
encodedreal     dq      3f00000000000000R
constantexp     dq      4 * 3
empty           dq      ?
multiple        dq      1, 2, 3, '$'
duplicate       dq      10 dup(?)
high_quad       dq      18446744073709551615
```

DT Directive

Syntax

[*name*] DT *initial-value*,,,

Description

The **DT** directive allocates and initializes 10 bytes of storage for each given *initial-value*.. An *initial-value* can be:

- An integer expression
- A packed decimal
- A 1- or 2-character string constant
- An encoded real number
- A **DUP** operator
- A question mark (?).

The question mark (?) represents an undefined initial value. If you give two or more expressions, separate them by commas (,).

The *name* is optional. If you give a *name*, the directive creates a variable of type **TBYTE** whose offset value is the current location counter value.

String constants must not exceed 2 characters in length. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are 2 in the string) is placed in the next byte. Zeros are placed in all remaining bytes.

Note: The **DT** directive assumes that constants with only decimal digits are packed decimals, not integers. For integer expressions, suffix the integer with a valid radix character. For example:

```
packeddecimal   dt      1234567890
integer         dt      16728D
character       dt      'a'
string          dt      'bc'
real            dt      1.5
encodedreal     dt      3f000000000000000000R
empty           dt      ?
multiple        dt      1, 2, 3, '$'
duplicate       dt      10 dup(?)
high_tbyte      dt      1208925819614629174706175D
```

DUP Operator

Syntax

count `DUP(`*initial-value,,,* `)`

Description

The **DUP** operator is a special operator that you can use with the Data Declaration and other directives to specify multiple occurrences of one or more initial values. The *count* defines the number of times to repeat the *initial-value.* An *initial value* can be any expression that evaluates to an integer value, a character constant, or another **DUP** operator. If you give more than one initial value, separate the values by commas (,). You can nest **DUP** operators up to 17 levels. For example:

```
DB      100 DUP(1)
```

This example generates 100 bytes with value 1.

```
DW      20 DUP( 1,2,3,4 )
```

This example generates 80 words of data. The first four words have the values 1, 2, 3, and 4, respectively. This pattern is duplicated for the remaining words.

```
DB      5 DUP(5 DUP(5 DUP (1)))
```

This example generates 125 bytes of data, with each byte having the value 1.

```
DD      14 DUP(?)
```

This example generates 14 doublewords of uninitialized data.

Symbol Declarations

The ***symbol declaration directives*** enable you to create and use symbols. A symbol is a descriptive name that represents a number, text, an instruction, or an address. Symbols make programs easier to read and maintain by letting descriptive names represent values. You can use a symbol anywhere the assembler allows its corresponding value.

Here are the symbol declaration directives:

=	Assign Absolutes
EQU	Equate Absolutes, Aliases, or Text Symbols
LABEL	Instruction or Data Labels

The following sections describe these directives in detail.

= Directive

Syntax

name = *expression*

Description

The = directive creates an absolute symbol by assigning the numeric value of *expression* to *name*. An absolute symbol is a name that represents a 16-bit value. The assembler does not allocate storage for the number. Instead, the assembler replaces each subsequent occurrence of the *name* with the value of the given *expression*.

The *expression* can be an integer, a 1- or 2-character string constant, a constant expression, or an address expression. Its value must not exceed 65,535. The *name* must be either a unique name or a name that was previously defined using the = directive.

Absolute symbols can be redefined at any time. For example:

```
integer         =       16728
string          =       'ab'
constantexp     =       3 * 4
addressexp      =       string
```

EQU Directive

Syntax

name EQU *expression*

Description

The **EQU** directive creates absolute symbols, aliases, or text symbols by assigning the *expression* to the given *name*. An ***absolute symbol*** is a name that represents a 16-bit value, an ***alias*** is a name that represents another symbol, and a ***text symbol*** is a name that represents a character string or other combination of characters. The assembler replaces each subsequent occurrence of the *name* with either the text or the value of the *expression,* depending on the type of expression given.

The *name* must be a unique name, not previously defined. The *expression* can be:

- An integer
- A string constant
- A real number
- An encoded real number
- An instruction mnemonic
- A constant expression
- An address expression.

Expressions that evaluate to integer values in the range 0 to 65,535 create absolute symbols and cause the assembler to replace the name with a value. All other expressions cause the assembler to replace the name with text.

You can use the **EQU** directive to create simple macros. Note that the assembler replaces a name with text before attempting to assemble the statement containing the name. Symbols defined using EQU directive cannot be redefined. For example:

```
integer         equu    16728         ; replaced with value
real            equ     3.14159       ; replaced with text
constantexp     equ     3 * 4         ; replaced with value
memoryop        equ     [bp]          ; replaced with text
mnemonic        equ     mov           ; replaced with text
addressexp      equ     real          ; replaced with text
string          equ     'Type Enter'  ; replaced with text
```

LABEL Directive

Syntax

name LABEL *type*

Description

The **LABEL** directive creates a new variable or label by assigning the current location counter value and the given *type* to *name* .

The *name* must be unique and not previously defined. The *type* can be any one of the following:

BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR

The *type* also can be the name of a valid structure type. For example:

```
subroutine        label   far
barray            label   byte
```

Type Declarations

The ***type declaration directives*** let you define data types that you then can use to create program variables consisting of multiple elements or fields. The directives associate one or more named fields with a given type name. You then can use the type name in a data declaration to create a variable of the given type.

Here are the type declaration directives:

STRUC and ENDS	Structure Types
RECORD	Record Types

The following sections describe these directives in detail.

STRUC and ENDS Directives

Syntax

name `STRUC`
    field-definitions
name `ENDS`

Description

The **STRUC** and **ENDS** directives mark the beginning and end of a type definition for a structure. Structure type definitions define the name of a structure type and the number, type, and default value of the fields contained in the type. Once the structure is defined, you can use structure types to declare structure variables.

The *name* defines the new name of the structure type. It must be unique. The *field-definitions* define the structure's fields. You can give any number of field definitions. The definitions must have these forms:

[*name*] DB *default-value*,,,
[*name*] DW *default-value*,,,
[*name*] DD *default-value*,,,
[*name*] DQ *default-value*,,,
[*name*] DT *default-value*,,,

The optional *name* defines the field name. The **DB**, **DW**, **DD**, **DQ**, and **DT** directives define the size of each field. The *default-value* defines the value to be given to the field if no initial value is given when the structure variable is declared. The *name* must be unique, and once defined, represents the offset from the beginning of the structure to the corresponding field. The *default-value* can define a number, a character or a string constant, or a symbol. It can also contain the **DUP** operator to define multiple values for the field. If the *default-value* is a string constant, the field has the same number of bytes as characters in the string. If multiple default values are given, they must be separated by commas.

A structure type definition can contain field definitions and comments only. It must not contain any other statements. This means structures cannot be nested. For example:

```
table   struc
        count   db      10
        value   dw      10 DUP(?)
        name    db      'font3'
table   ends
```

In this example, the fields are **count**, **value**, and **name**.

- The **count** field is a single byte value initialized to **10**.
- The **value** is an array of **10** uninitialized word values.
- The **name** is a character array of 5 bytes initialized to **font3**.

The field names **count**, **value**, and **name** have the offset values 0, 1, and 21, respectively.

RECORD Directive

Syntax

recordname `RECORD` *fieldname*`:`*width* `[=`*exp*`]` *,,,*

Description

The **RECORD** directive defines a record type for an 8- or 16-bit record that contains one or more fields.

- The *recordname* is the name of the record type to be used when creating the record.
- The *fieldname* is the name of a field in the record.
- The *width* is the number of bits in the field.
- The *exp* is the initial (or default) value for the field.

You can give any number of *fieldname:width = exp* combinations with the record, as long as you separate each from the preceding one by a comma (,). The sum of the widths for all fields must not exceed 16.

The *width* must be a constant in the range 1 to 16. If the total width of all declared fields is larger than 8 bits, the assembler uses 2 bytes. Otherwise, the assembler uses only 1 byte.

If you supply *=exp* , the assembler defines the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character for *exp*. The *exp* must not contain a forward reference to any symbol.

In all cases, the first field you declare goes into the most significant bits of the record. The assembler places successive declared fields in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the assembler shifts the entire record right so that the last bit of the last field is the lowest bit of the record. Unused bits will be in the high end of the record. For example:

```
encode RECORD     high:4, mid:3, low:3
```

This example creates a record type **encode** having three fields: **high**, **mid**, and **low**. The record occupies 16 bits of memory. The **high** field is in bits 6 to 9, **mid** in bits 3 to 5, and **low** in bits 0 to 2. The remaining high-order bits are unused.

```
item   RECORD     char:7='Q', weight:4=2
```

This example creates a record type **item** having two fields: **char** and **weight**. These values are initialized to the letter Q and the number 2, respectively.

Structure and Record Declarations

Structure declarations and ***record declarations*** let you generate a block of data bytes that have many elements or fields. A structure or record declaration consists of a previously-defined structure or record type name and a set of initial values.

The following sections describe these declarations in detail.

Structure Declarations

Syntax

[*name*] *strucname* < [*initial-value*,,,]>

Description

A ***structure variable*** is a variable that has one or more fields of different sizes.

- The *name* is the name of the variable.
- The *strucname* is the name of a structure type that has been created using the STRUC directive.
- The *initial-value* is one or more values defining the initial value of the structure.

The *name* is optional. If the name is not given, the assembler allocates space for the structure, but does not create a name that you can use to access the structure.

The *initial-value* can be an integer, a string constant, or an expression that evaluates to a value having the same type as the corresponding field. The angle brackets (< >) are required even if you do not give an initial value. One *initial-value* can be given for each field in the structure. If you give more than one initial value, separate the values with commas (,).

If you use the **DUP** operator, you only need to enclose the values within the parentheses in angle brackets. You do not have to initialize all fields in a structure. If you leave an initial value blank, the assembler automatically uses the default initial value of the field. This is defined by the structure type. If there is no default value, the field is uninitialized. If the overriding string is shorter than the initial string, the assembler pads the space to the right with blanks.

Note: You cannot initialize any structure field that has multiple values if this field was given a default initial value when the structure was defined. For example:

```
struct1 table    <>
```

This example creates a structure variable named `struct1` whose type is given by the structure type `table`. The initial values of the fields in the structure are set to the default values for the structure type, if any.

```
struct2 table   <0,,>
```

This example creates a structure variable named `struct2`. Its type is also `table`. The initial value for the first field is set to 0. The default values defined by the structure type are used for the remaining two fields.

```
struct3 table   10 DUP(<0,,>)
```

This example creates a variable `struct3` containing 10 structures of the type `table`. The first field in each structure is set to the initial value zero. All remaining fields receive the default values.

Record Declarations

Syntax

[*name*] *recordname* < [*initial-value*,,,]>

Description

A record variable is an 8- or 16-bit value whose bits are divided into one or more fields.

- The *name* is the name of the variable.
- The *recordname* is the name of a record type that has been created using the **RECORD** directive.
- The *initial-value* is one or more values defining the initial value of the record.

One *initial-value* can be given for each field in the record. This value can be an integer, a string constant, or any expression that evaluates to a value that is no larger than can be represented in the specified field width. Angle brackets (< >) are required even if no initial value is given. If you give more than one initial value, separate the values with commas. If you use the **DUP** operator, you only need to enclose the values within the parentheses in angle brackets.

The *name* is optional. If the name is not given, the assembler allocates space for the record but does not create a variable that you can use to access the record.

You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is defined by the record type. If there is no default value, the field is uninitialized. For example:

```
rec1    encode  <>
```

This example creates a record variable named `rec1` whose type is given by the record type `encode`. The initial values of the fields in the record are set to the default values for the record type, if any.

```
table   item    10 DUP(<'A',2>)
```

This example creates a variable `table` containing 10 records of the record type `item`. The fields in these records are all set to the initial values A and 2.

```
passkey encode  <,,7>
```

This example creates a record variable named `passkey`. Its type is `encode`. The initial values for the first two fields are the default values defined by the record type. The initial value for the third field is 7.

Chapter 5. Operands and Expressions

Introduction

This chapter describes the syntax and meaning of operands and expressions you can use in assembly language statements and directives. ***Operands*** represent values, registers, or memory locations to be acted on by instructions or directives. ***Expressions*** are combinations of operands and arithmetic, logical, bitwise, and attribute operators. An expression evaluates to a value or memory location to be acted on by an instruction or a directive.

Operands

An operand is a constant, a label, a variable, or another symbol that you use in an instruction or directive to represent a value, a register, or a memory location to be acted on. Operand types include:

Constant
Direct Memory
Relocatable
Location Counter
Register
Based
Indexed
Based Indexed
Structure
Record
Record Field

Constant Operands

Syntax

number|*string*|*expression*

Description

A ***constant operand*** is a number, a string constant, a symbol, or an expression that evaluates to a fixed value. Constant operands, unlike other operands, represent values to be acted on rather than memory addresses. For example:

```
mov     ax,9
mov     al,'c'
mov     bx,65535/3
mov     cx,count
```

Direct Memory Operands

Syntax

segment:*offset*

Description

A ***direct memory operand*** is a pair of segment and offset values that represent the absolute memory address of one or more bytes of memory. The *segment* can be a segment register name (**CS**, **DS**, **SS**, or **ES**), a segment name, or a group name. The *offset* must be an integer, an absolute symbol, or an expression that evaluates to a value within the range 0 to 65,535. For example:

```
mov     dx, ss:0031H
mov     bx, DATA:0
mov     cx, DGROUP:block
```

Relocatable Operands

syntax

symbol

Description

A ***relocatable operand*** is any symbol that represents the memory address (segment and offset) of an instruction or data to be acted on. Relocatable operands, unlike direct memory operands, are relative to the start of the segment or group in which the symbol is defined. They have no explicit value until the program has been linked. For example:

```
call    main
mov     bx,local
mov     bx,offset DGROUP:table
```

Location Counter Operands

Syntax

`$`

Description

The ***location counter*** is a special operand that, during assembly, represents the current location within the current segment. The location counter has the same attributes as a **NEAR** label. It represents an instruction address that is relative to the current segment. Its offset is equal to the number of bytes that have been generated for that segment to that point. After each statement in the segment has been assembled, the assembler increments the offset by the number of bytes generated. For example:

```
target  equ     $
        mov     ax,1
        .
        .
        .
        jmp     target
```

Register Operands

Syntax

reg-name

Description

A ***register operand*** is the name of a processor register. Register operands direct instructions to carry out actions on the contents of the given registers. The *reg-name* can be any one of the following:

```
ax      ah      al      bx      bh      bl
cx      ch      cl      dx      dh      dl
cs      ds      ss      es      sp      bp
di      si
```

You can use any combination of uppercase and lowercase letters.

The **ax**, **bx**, **cx**, and **dx** registers are 16-bit general purpose registers that you can use for any data or numeric manipulation. The **ah**, **bh**, **ch**, and **dh** registers represent the high 8 bits of the corresponding general purpose registers. Similarly, **al**, **bl**, **cl**, and **dl** represent the low-order 8 bits of the general purpose registers.

Here are the segment registers and the addresses they contain:

cs	The code
ds	The data
ss	The stack
es	The extra segments.

All instruction and data addresses are relative to the segment address in one of these registers.

The **sp** register is the 16-bit stack pointer register. The stack pointer contains the current top of stack address. This address is relative to the segment address in the **ss** register and is automatically modified by instructions that access the stack.

The **bx**, **bp**, **di**, and **si** registers are 16-bit base and index registers. These are general purpose registers that are typically used for pointers to program data.

The 16-bit flag register contains twelve flags whose positions and meaning are defined in the following table:

Flag Bit	Meaning
0	Carry flag
2	Parity flag
4	Auxiliary flag
6	Zero flag
7	Sign flag
8	Trap flag
9	Interrupt-enable flag
10	Direction flag
11	Overflow flag
12	IOPL flag (low order bit)
13	IOPL flag (high order bit)
14	Nested task flag

You can access the contents of the flags register using the **LAHF**, **SAHF**, **PUSHF**, and **POPF** instructions.

Based Operands

Syntax

disp`[bp]`
disp`[bx]`

Description

A ***based operand*** represents a memory address relative to one of the base registers: **bp** or **bx** . The *disp* can be any immediate or direct memory operand. It must evaluate to an absolute number or memory address. If no *disp* is given, zero is assumed.

The effective address of a based operand is the sum of the *disp* value and the contents of the given register. If you use **bp**, the operand's address is relative to the segment pointed to by the **ss** register. If you use **bx**, the address is relative to the segment pointed to by the **ds** register.

Based operands have a variety of forms. The following illustrate a few of these forms:

`[`*disp*`] [bp]`
`[bp+`*disp*`]`
`[bp].`*disp*
`[bp]+`*disp*

In each case, the effective address is the sum of *disp* and the contents of the given register. For example:

```
mov     ax,[bp]
mov     ax,[bx]
mov     ax,12[bx]
mov     ax,fred[bp]
```

Indexed Operands

Syntax

disp`[si]`
disp`[di]`

Description

An ***indexed operand*** represents a memory address that is relative to one of the index registers, **si** or **di**. The *disp* can be any immediate or direct memory operand. It must evaluate to an absolute number or a memory address. If no *disp* is given, zero is assumed.

The effective address of an indexed operand is the sum of the *disp* value and the contents of the given register. The address is always relative to the segment pointed to by the **ds** register.

Indexed operands have a variety of alternate forms. Here are a few of these forms:

[*disp*] [di]
[di +*disp*]
[di].*disp*
[di]+*disp*

In each case, the effective address is the sum of *disp* and the contents of the given register. For example:

```
mov     ax,[si]
mov     ax,[di]
mov     ax,12[di]
mov     ax,fred[si]
```

Based Indexed Operands

Syntax

disp [bp][si]
disp [bp][di]
disp [bx][si]
disp [bx][di]

Description

A ***based indexed operand*** represents a memory address that is relative to a combination of base and index registers. The *disp* can be any immediate or direct memory operand. It must evaluate to an absolute number or memory address. If no *disp* is given, zero is assumed.

The effective address of a based indexed operand is the sum of the *disp* value and the contents of the given registers. If you use the **bp** register, the address is relative to the segment pointed to by the **ss** register. Otherwise, the address is relative to the segment pointed to by the **ds** register.

Based indexed operands have a variety of alternate forms. Here are a few of these forms:

[*disp*][bp][di]
[bp+di+*disp*]
[bp+di].*disp*
[di]+*disp*+[bp]

In each case, the effective address is the sum of *disp* and the contents of the given registers. For example:

```
mov      ax,[bp][si]
mov      ax,[bx + di]
mov      ax,12[bp + di]
mov      ax,fred[bx][si]
```

Structure Operands

Syntax

variable.field

Description

A ***structure operand*** represents the memory address of one member of a structure. The *variable* must be the name of a structure or must be a memory operand that resolves to the address of a structure. The *field* must be the name of a field within that structure.

The effective address of a structure operand is the sum of the offsets of *variable* and *field*. The address is relative to the segment or group in which the *variable* is defined.

In the following examples, **current_date** is assumed to be the structure defined by the following:

```
date     struc
         month    dw  ?
         day      dw  ?
         year     dw  ?
date     ends
current_date   date <'ja','01','84'>
```

For example:

```
mov      ax,current_date.day
mov      current_date.year, '85'
```

You can use structure operands to access values on the stack. One method to do this is to copy the current stack address into the **bp** register. Then, use **[bp].member** to access elements on the stack. This method makes all values on the stack available in any format you desire.

Record Operands

Syntax

recordname < [*value*],,, >

Description

A ***record operand*** refers to the value of a record type. The operands can be in expressions. The *recordname* must be the name of a record type defined in the source file. The optional *value* is the value of a field in the record. If you give more than one *value*, separate the values by commas (,). The enclosing angle brackets are required, even if no *value* is given. If you do not give a *value* for a field, the default value for that field is used. For example:

```
mov     ax,encode <1,3,2>
mov     cx,key <,7>
```

Record Field Operands

Syntax

record-fieldname

Description

The ***record field operand*** represents the location of a field in its corresponding record. The operand evaluates to the bit position of the low-order bit in the field and can be used as a constant operand.

The *record-fieldname* must be the name of a previously defined record field. It can be used to extract or insert data into a record. When the *record-fieldname* is referenced in other than the **MASK** or **WIDTH** operator, its value is the shift count required to shift it to the low-order bit position. For additional uses of the *record-fieldname*, refer to "WIDTH Operator" on page 5-19 and "MASK Operator" on page 5-20.

In the following examples, assume that the record `rec1` is defined as:

```
rtype   RECORD          field1:3,field2:6,field3:7
rec1    rtype  <111b, 010010b, 1111111b >
```

For example:

```
mov     ax,field1
```

This example moves the shift count for `field1` into `ax`. The `ax` register now contains a value of 13..

```
mov     dx,rec1
and     dx,mask field2
mov     cl,field2
shr     dx,cl
```

This example illustrates the usage of the shift count and mask operator to store the value of the second field of `rec1`, right-justified, in register `dx`. First, the value of `rec1` is moved into register `dx`. Next, `dx` is and'ed with `mask field2` (which contains a value of 1F80H). This results in retaining only the second field (bits 7-12) in the register `dx`. Next, the shift count of `field2` (7) is moved into register `cl`. Finally, `dx` is shifted right 7 bits. The second field of `rec1` is now stored right-justified in register `dx`. Register `dx` now has a value of 0012H.

Expressions

An expression is a combination of operands and operators that evaluates to a single value. Operands in expressions can be any of the those described in this chapter. The result of an expression can be a value or a memory location, depending on the types of operands and operators used.

The assembler provides a variety of operators. Arithmetic, shift, relational, and bitwise operators manipulate and compare the values of operands. Attribute operators manipulate the attributes of operands, such as their type, address, and size. The following sections describe the operators in detail.

Arithmetic Operators

Syntax

exp1 * *exp2*
exp1 / *exp2*
exp1 MOD *exp2*
exp1 + *exp2*
exp1 - *exp2*
+ *exp*
- *exp*

Description

Arithmetic operators provide the common mathematical operations. The operators have the following meanings:

Operator	**Meaning**
*	Multiplication
/	Integer division
MOD	Remainder after division (modulus)
+	Addition
-	Subtraction
+	Positive (unary)
-	Negative (unary).

For all arithmetic operators except + and -, the expressions *exp1* and *exp2* must be integer numbers. You can use the + operator to add an integer number to a relocatable memory operand. You can use the - operator to subtract an integer number from a relocatable memory operand. You also can use the - operator to subtract one relocatable operand from another, but only if the operands refer to locations within the same segment. The result is an absolute value. For example:

```
14 * 4              ; equals  56
14 / 4              ; equals  3
14 MOD 4            ; equals  2
14 + 4              ; equals  18
14 - 4              ; equals  10
14 - +4             ; equals  10
14 - -4             ; equals  18
alpha + 5           ; add 5 to alpha's offset
alpha - 5           ; subtract 5 from alpha's offset
alpha - beta        ; subtract beta's offset from alpha's
```

SHR and SHL Operators

Syntax

expression SHR *count*
expression SHL *count*

Description

The **SHR** and **SHL** operators shift the given *expression* right or left by *count* number of bits. Bits shifted off the end of the expression are lost. If *count* is greater than or equal to 16, the result is 0. For example:

```
01110111B SHL 3           ; equals 10111000B
01110111B SHR 3           ; equals 00001110B
```

Relational Operators

Syntax

exp1 EQ *exp2*
exp1 NE *exp2*
exp1 LT *exp2*
exp1 LE *exp2*
exp1 GT *exp2*
exp1 GE *exp2*

Description

The ***relational operators*** compare the expressions *exp1* and *exp2* and return true (0FFFFH) if the given condition is satisfied or false (0000H) if it is not. The expressions must resolve to absolute values. The operators have the following meanings:

Operator	Condition is satisfied when:
EQ	Operands are equal.
NE	Operands are not equal.
LT	Left operand is less than right.
LE	Left operand is less than or equal to right.
GT	Left operand is greater than right.
GE	Left operand is greater than or equal to right.

Use relational operators with conditional directives and conditional instructions to direct program control. For example:

```
1   EQ   0          ; false
1   NE   0          ; true
1   LT   0          ; false
1   LE   0          ; false
1   GT   0          ; true
1   GE   0          ; true
```

Bitwise Operators

Syntax

NOT *exp*
exp1 AND *exp2*
exp1 OR *exp2*
exp1 XOR *exp2*

Description

The ***logical operators*** perform bitwise operations on the given expressions. In a bitwise operation, the operation is performed on each bit in an expression rather than on the expression as a whole. The expressions must resolve to absolute values.

The operators have the following meanings:

Operator	**Meaning**
NOT	Inverse
AND	Boolean AND
OR	Boolean OR
XOR	Boolean exclusive OR

For example:

```
NOT 11110000B                ; equals 00001111B
01010101B AND 11110000B      ; equals 01010000B
01010101B OR 11110000B       ; equals 11110101B
01010101B XOR 11110000B      ; equals 10100101B
```

Index Operator

Syntax

expression1 [*expression2*]

Description

The ***index operator***, **[ ]**, adds the value of *expression1* to *expression2*. This operator is identical to the + operator, except that *expression1* is optional.

If you give *expression1*, the expression must appear to the left of the operator. It can be any integer value, absolute symbol, or relocatable operand. If you do not give *expression1*, the integer value, 0, is assumed. If *expression1* is a relocatable operand, *expression2* must be an integer value or absolute symbol. Otherwise, *expression2* can be any integer value, absolute symbol, or relocatable operand.

You typically use the index operator to index elements of an array, such as individual characters in a character string. For example:

```
mov     al,string[3]
mov     ax,array[4]
mov     string[LAST],al
```

PTR Operator

Syntax

type PTR *expression*

Description

The **PTR** operator treats the variable or the label given by the *expression* as a variable or a label having the type given by *type*. The *type* must be one of the following names or values:

BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFFh
FAR	0FFFEh

The *expression* can be any operand. Use the **BYTE**, **WORD**, and **DWORD** types with memory operands only. Use the **NEAR** and **FAR** types with labels only.

You typically use the **PTR** operator with forward references to explicitly define what size or distance a reference has. If the **PTR** operator is not used, the assembler assumes a default size or distance for the reference. You also can use the **PTR** operator to give instructions access to variables in ways that would otherwise generate errors, for example, accessing the high-order byte of a **WORD** size variable. For example:

```
call    far ptr subrout3
mov     byte ptr [array],1
add     al,byte ptr [full_word]
```

Segment Override Operator

Syntax

segment-register : *expression*
segment-name : *expression*
group-name : *expression*

Description

The ***segment override operator*** (**:**) forces the assembler to compute the address of a given variable or label using the beginning of the given *segment-register*, *segment-name*, or *group-name*. If you give a *segment-name* or *group-name*, you must assign the name to a segment register with a previous **ASSUME** directive and define it using a **SEGMENT** or a **GROUP** directive. The *expression* can be an absolute symbol or relocatable operand. The *segment-register* must be one of **CS**, **DS**, **SS**, or **ES**.

By default, the effective address of a memory operand is computed relative to the **DS**, **SS**, or **ES** register, depending on the instruction and operand type. Similarly, the assembler assumes all labels to be **NEAR**. You can override these default types using the segment override operator. For example:

```
mov     ax,es:[bx][si]
mov     _TEXT:far_label,ax
mov     ax,DGROUP:variable
mov     al,cs:0001H
```

SHORT Operator

Syntax

`SHORT` *label*

Description

The **SHORT** operator sets the type of the given *label* to **SHORT**. You can use short labels in jump instructions whenever the distance from the label to the instruction is not more than 127 bytes. Instructions using short labels are one byte smaller than identical instructions using near labels. For example:

```
jmp     short repeat
```

THIS Operator

Syntax

`THIS` *type*

Description

The **THIS** operator creates an operand whose offset and segment value are equal to the current location counter value and whose type is given by *type*. The *type* can be any one of the following:

NEAR
FAR
BYTE
WORD
DWORD
QWORD
TBYTE

You typically use the **THIS** operator with the **EQU** or = directive to create labels and variables. This is similar to using the **LABEL** directive to create labels and variables. For example:

```
tag     equu    this byte
```

This example is equivalent to the statement **TAG LABEL BYTE**.

```
check   =       this near
```

This example is equivalent to the statement **CHECK LABEL NEAR**.

HIGH and LOW Operators

Syntax

`HIGH` *expression*
`LOW` *expression*

Description

The **HIGH** and **LOW** operators return the high and low 8 bits of the given *expression* . The **HIGH** operator returns the high-order 8 bits of the *expression*. The **LOW** operator returns the low-order 8 bits. The *expression* can be any value. For example:

```
mov     ah,high word_value
mov     al,low OFFFFH
```

SEG Operator

Syntax

`SEG` *expression*

Description

The **SEG** operator returns the segment value of the given *expression*. The *expression* can be any label, variable, segment name, group name, or other symbol. For example:

```
mov     ax,seg variable_name
mov     ax,seg label_name
```

OFFSET Operator

Syntax

`OFFSET` *expression*

Description

The **OFFSET** operator returns the offset of the given *expression*. The *expression* can be any label, variable, segment name, or other symbol. The returned value is the number of bytes between the item and the beginning of the segment in which it is defined. For a segment name, the return value is the offset from the start of the segment to the most recent byte generated for that segment.

You can use the segment override operator (:) to force **OFFSET** to return the number of bytes between the item in the *expression* and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group; it is illustrated in the second line below. For example:

```
mov     bx,offset subrout3
mov     bx,offset DGROUP:array
```

The returned value is always a relative value that is subject to change by the linker when the program is actually linked.

TYPE Operator

Syntax

`TYPE` *expression*

Description

The **TYPE** operator returns a number representing the type of the given *expression.* If the *expression* is a variable, the operator returns the size of the operand in bytes. If the *expression* is a label, the operator returns 0FFFFH if the label is **NEAR** and 0FFFEH if the label is **FAR**. Note that you can use the return value to specify the type for a **PTR** operator, as is illustrated in the second line below. For example:

```
mov     ax,type array
jmp     (type get_loc) ptr destiny
```

.TYPE Operator

Syntax

`.TYPE` *expression*

Description

The **.TYPE** operator returns a byte that defines the mode and scope of the given *expression.* If the *expression* is not valid, **.TYPE** returns 0.

The variable's attributes are returned in bits 0, 1, 5, and 7 as follows:

Bit Position	**If Bit=0**	**If Bit=1**
0	Absolute	Program related
1	Not data related	Data related
5	Not defined	Defined
7	Local scope	External scope

If both the scope bit and defined bit are 0, the *expression* is not valid.

You typically use the **.TYPE** operator with conditional directives where an argument may need to be tested to make a decision regarding program flow. For example:

```
x        db       12
z        equ      .type x
```

This example sets `z` to 34 because `x` is a defined data related expression.

LENGTH Operator

Syntax

`LENGTH` *variable*

Description

The **LENGTH** operator returns the number of **BYTE**, **WORD**, **DWORD**, **QWORD**, or **TBYTE** elements in the given *variable*. The size of each element depends on the defined type of the *variable*.

Only variables that have been defined using the **DUP** operator return values greater than one. The return value is always the number that precedes the first **DUP** operator.

In the following examples, assume the definitions:

```
array    dw       100    dup(1)
table    dw       100    dup(1,10 dup(?))
```

For example:

```
mov      cx,length array
```

In this example, **LENGTH** returns 100.

```
mov      cx,length table
```

In this example, **LENGTH** returns 100. The return value does not depend on any nested **DUP** operators.

SIZE Operator

Syntax

`SIZE` *variable*

Description

The **SIZE** operator returns the total number of bytes allocated for the given *variable*. The return value is equal to the return value of **LENGTH** times the return value of **TYPE**. In the following example, assume the definition:

```
array    dw        100     dup(1)
```

For example:

```
mov      bx,size array
```

In this example, **SIZE** returns 200.

WIDTH Operator

Syntax

`WIDTH` *record-fieldname* ¦ *record*

Description

The **WIDTH** operator returns the width (in bits) of the given record field or record. The *record-fieldname* must be the name of a record defined in a field. The *record* must be the name of a record. In the following example, assume that the record `rec1` is defined as:

```
rtype    RECORD            field1:3,field2:6,field3:7
rec1     rtype   <>
```

For example:

```
WIDTH field1     ;equals 3
WIDTH field2     ;equals 6
WIDTH field3     ;equals 7
WIDTH rtype      ;equals 16
```

MASK Operator

Syntax

`MASK` *record-fieldname* | *record*

Description

The **MASK** operator returns a bit mask for the bit positions in a record occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a record bit. All other bits contain 0. The *record-fieldname* must be the name of a record field.

In the following example, assume that the record `rec1` is defined as:

```
rtype   RECORD          field1:3,field2:6,field3:7
rec1    rtype   <>
```

For example:

```
MASK field1      ;equals E000H
MASK field2      ;equals 1F80H
MASK field3      ;equals 007FH
MASK rtype       ;equals 0FFFFH
```

Expression Evaluation and Precedence

The assembler evaluates expressions according to the rules of operator precedence and order. The assembler performs operations of highest precedence first. It performs operations of equal precedence from left to right. You can override this default order of evaluation using enclosing parentheses. The assembler always performs operations in parentheses before any adjacent operations.

The following table lists the precedence of all operators. Operators on the same line have equal precedence.

Precedence	Operators
Highest	
1	LENGTH, SIZE, WIDTH, MASK
2	( )
3	[ ]
4	:
5	PTR, OFFSET, SEG, TYPE, THIS
6	HIGH, LOW

7	*, /, MOD, SHL, SHR
8	+, -
9	EQ, NE, LT, LE, GT, GE
10	NOT
11	AND
12	OR, XOR
13	SHORT, .TYPE
Lowest	

For example:

```
8 / 4 * 2                   ; equals 4
8 / (4 * 2)                 ; equals 1
8 + 4 * 2                   ; equals 16
(8 + 4) * 2                 ; equals 24
8 EQ 4 AND 2 LT 3           ; equals 0000H (false)
8 EQ 4 OR 2 LT 3            ; equals 0FFFFH (true)
```

Forward References

Although the assembler permits forward references to labels, variable names, segment names, and other symbols, such references can lead to assembly errors if not used properly. A ***forward reference*** is any use of a name before it has been formally declared. For example, in the **JMP** instruction below, the label `target` is a forward reference:

```
        jmp     target
        mov     ax,0
target:
```

Whenever the assembler encounters an undefined name in pass 1, it assumes that the name is a forward reference. If only a name is given, the assembler makes assumptions about that name's type and segment register and uses these assumptions to generate code or data for the statement. For example, in the **JMP** instruction above, the assembler assumes that `target` is an instruction label having **NEAR** type. It generates 3 bytes of instruction code for the instruction.

The assembler bases its assumptions on the statement containing the forward reference. Errors can occur when these assumptions are incorrect. For example, if `target` really was a **FAR** label and not a **NEAR** label, the assumption made by the assembler in pass 1 would cause a phase error. In other words, the assembler would generate 5 bytes of instruction code for the **JMP** instruction in pass 2 but only 3 in pass 1.

To avoid errors with forward references, use the segment override (:), **PTR**, and **SHORT** operators to override the assumptions made by the assembler whenever necessary. The following guidelines list when to use these operators.

- If a forward reference is a variable that is relative to the **ES**, **SS**, or **CS** register, use the segment override operator (:) to specify the variable's segment register, segment, or group. For example:

  ```
  mov     ax,ss:stacktop
  inc     data:time[1]
  add     ax,dgroup:_I
  ```

  If you do not use the segment override operator, the assembler assumes that the variable is referenced relative to the **DS** register.

- If a forward reference is an instruction label in a **JMP** instruction, use the **SHORT** operator if the instruction is less than 128 bytes from the point of reference. For example:

  ```
  jmp     short target
  ```

  If you do not use **SHORT**, the assembler assumes that the instruction is greater than 128 bytes away. This does not cause an error, but it does cause the assembler to generate an extra **NOP** instruction that is not needed.

- If a forward reference is an instruction label in a **CALL** or a **JMP** instruction, use the **PTR** operator to specify the label's type. For example:

  ```
  call    far ptr print
  jmp     near ptr exit
  ```

  The assembler assumes that the label has **NEAR** type, so **PTR** need not be used for **NEAR** labels. If the label has **FAR** type, however, and you do not use **PTR**, a phase error results.

- If the forward reference is a segment name with a segment override operator, use the **GROUP** statement to associate the segment name with a group name. Then use the **ASSUME** statement to associate the group name with a segment register. For example:

```
dgroup  segment stack
        assume  ss:dgroup

code    segment
        .
        .
        .
        mov     ax,stack:stacktop
        .
        .
        .
```

If you do not associate a group with the segment name, the assembler may ignore the segment override and use the default segment register for the variable. This usually results in a phase error in pass 2.

Strong Typing for Memory Operands

The assembler carries out strict syntax checks for all instruction statements, including strong typing for operands that refer to memory locations. This means that any relocatable operand you use in an instruction that operates on an implied data type must either have that type or have an explicit type override (**PTR** operator).

For example, in the following program segment, the variable `string` is incorrectly used in a move instruction:

```
string  db      "A message."

        mov     ax,string[1]
```

This statement results in an `Operand types must match` error because `string` has **BYTE** type and the instruction expects a variable having **WORD** type.

To avoid this error, use the **PTR** operator to override the variable's type. The statement:

```
        mov     ax,WORD PTR string[1]
```

assembles correctly and executes as expected.

Chapter 6. Global Declarations

Introduction

The ***global declaration directives*** let you define labels, variables, and absolute symbols that can be accessed globally; that is, from all modules in a program. Global declarations transform local symbols (labels, variables, and other symbols that can be used only in the source files in which they are defined) into global symbols that are available to all other modules.

The global declaration directives include:

PUBLIC
EXTRN

You use the **PUBLIC** directive in public declarations. A ***public declaration*** transforms a locally defined symbol into a global symbol, making it available to other modules. You use the **EXTRN** directive in external declarations. An ***external declaration*** makes a global symbol's name and type known in a source file, letting the global symbol be used in that file. Every global symbol must have a public declaration in exactly one source file of the program. A global symbol can have external declarations in any number of other source files. The following sections describe the global declaration directives in detail.

PUBLIC Directive

Syntax

`PUBLIC` *name,,,*

Description

The **PUBLIC** directive makes the variable, the label, or the absolute symbol given by *name* available to all other modules in the program. The *name* must be the name of a variable, a label, or an absolute symbol defined within the current source file. Absolute symbols, if given, can only represent 1- or 2-byte integer or string values.

By default, the assembler preserves all lowercase names when writing them to the object file and assembly listing. For more information on the options available refer to the **-M** options in "Using the Options" on page 1-6. For example:

```
        public  true,test,start
true    =       OFFFFH
test    db      1
start   label   far
```

EXTRN Directive

Syntax

`EXTRN` *name*:*type,,,*

Description

The **EXTRN** directive defines an external variable, label, or symbol named *name* and whose type is *type*. An external item is any variable, label, or symbol that you have publicly declared in another module of the program.

The *name* must be the name of a variable, a label, or a symbol defined in another module of the program and listed in a **PUBLIC** directive of the corresponding source file. The *type* must match the type given to the item in its actual definition. It can be any one of the following:

BYTE	WORD
DWORD	QWORD
TBYTE	FAR
NEAR	ABS

The **ABS** type is reserved for symbols that represent absolute values.

Although the actual address is not determined until link time, the assembler may assume a default segment for the external item based on where you place the **EXTRN** directive in the module. If you place the directive inside a segment, the external item is assumed to be relative to that segment. In this case, the item's public declaration (in some other module) must be in a segment having the same name and attributes. If you place the directive outside all segments, no assumption is made about to what segment the item is relative. The item's public declaration can be in any segment in any module. In either case, you can use the segment override operator (:) to override an external variable's or label's default segment. For example:

```
extrn   tagn:near
extrn   var1:word,var2:dword
```

Program Example

The following source files illustrate a program that uses public and external declarations to access instruction labels.

The program consists of two modules, named `startm` and `printm`. The `startm` module is the program's main module. Execution starts at the instruction labeled `_main` in `startm`, and passes to the instruction labeled `_print` in `printm` where the PC XENIX **printf** routine is called to print the message `Hello` on the screen. Execution then returns to the instruction labeled `_finish` in `startm`.

Startm Module:

```
                 NAME    startm
                 public  _main, _finish
                 extrn   _exit:near
                 extrn   _print:near

_DATA   segment word public 'DATA'
_DATA   ends

DGROUP  group  _DATA

_TEXT   segment byte public 'CODE'
                 assume  cs:_TEXT, ds:DGROUP

_main:
                 jmp   _print
_finish:
                 push   0      ; exit(0)
                 call   _exit
_TEXT   ends
                 end
```

Printm Module:

```
                  NAME    printm
                  public  _print
                  extrn   _finish:near
                  extrn   _printf:near

_DATA   segment word public 'DATA'
string            db "Hello.", 10, 0
_DATA   ends

DGROUP group  _DATA

_TEXT   segment byte public 'CODE'
                  assume  cs:_TEXT, ds:_DGROUP
_print:
                  push offset DGROUP:string
                  call _printf
                  add sp, 2
                  jmp _finish
_TEXT   ends

                  end
```

In this example, `startm` publicly declares two symbols, `_main` and `_finish`, making the symbols available to the `printm` source file. Both of these symbols are locally defined as instruction labels in the `printm` source file. The `startm` file also contains an external declaration of the symbol `_print`. This declaration defines `_print` to be a **NEAR** label and is assumed to have been publicly declared in the `printm` source file. The label is used in a **JMP** instruction given later in the file.

The `printm` file contains a public declaration of the symbol `_print` and an external declaration of the symbols `_finish` and `printm`. In this case, `_print` is locally defined as a **NEAR** label and matches the external declaration given to it in `startm`. The symbol `_finish` is declared to be a **NEAR** label, matching its definition in `startm`. The symbol `_printf` references a C library routine (**printf**) to display the message `''Hello.''`.

Before you can execute this program, you must assemble these source files individually. Then link the files together as a small model program using the system linker.

Chapter 7. Conditional Assembly

Introduction

The ***conditional assembly directives*** provide conditional assembly of blocks of statements within a source file. The conditional assembly directives are:

IF
IFE
IF1
IF2
IFDEF
IFNDEF
IFB
IFNB
IFIDN
IFDIF
ELSE
ENDIF

You can use the **IF** directives and the **ENDIF** and **ELSE** directives to enclose the statements to be considered for conditional assembly. The conditional block takes the form:

```
IF
     statements
ELSE
     statements
ENDIF
```

The *statements* can be any valid statements, including other conditional blocks. The **ELSE** directive is optional.

The assembler assembles the statements in the conditional block only if the condition that satisfies the corresponding **IF** directive is met. If the conditional block contains an **ELSE** directive, however, the assembler assembles only the statements up to the **ELSE** directive. The statements following the **ELSE** directive are assembled only if the **IF** condition is not met. An **ENDIF** directive must mark the end of the conditional block. No more than one **ELSE** for each **IF** directive is allowed.

IF directives can be nested up to 255 levels. To avoid ambiguity, a nested ELSE directive always belongs to the nearest, preceding **IF** directive.

IF and IFE Directives

Syntax

`IF` *expression*
`IFE` *expression*

Description

The **IF** and **IFE** directives test the value of an *expression*. The **IF** directive grants assembly if *expression* is nonzero (true). The IFE directive grants assembly if *expression* is 0 (false). The *expression* must resolve to an absolute value and must not contain forward references. For example:

```
if  DEBUG
        extrn dump:far
        extrn trace:far
        extrn breakpoint:far
endif
```

IF1 and IF2 Directives

Syntax

```
IF1
IF2
```

Description

The **IF1** and **IF2** directives test the current assembly pass. The **IF1** directive grants assembly on pass 1 only. **IF2** grants assembly on pass 2. The directives take no arguments. For example:

```
if1
        %out Pass 1 Starting
endif
```

IFDEF and IFNDEF Directives

Syntax

`IFDEF` *name*
`IFNDEF` *name*

Description

The **IFDEF** and **IFNDEF** directives test whether or not the given *name* has been defined. The **IFDEF** directive grants assembly if *name* is a label, a variable, or a symbol. The **IFNDEF** directive grants assembly if you have not yet defined *name*.

The *name* can be any valid name. If *name* is a forward reference, the assembler considers it undefined on pass 1, but defined on pass 2. For example:

```
ifndef BUFFER
BUFFER       db       10 dup(?)
endif
```

IFB and IFNB Directives

Syntax

`IFB` <*arg*>
`IFNB` <*arg*>

Description

The **IFB** and **IFNB** directives test the given *arg*. The **IFB** directive grants assembly if *arg* is blank. The **IFNB** directive grants assembly if *arg* is not blank. The *arg* can be any name, number, or expression. You must use the angle brackets (< >).

Use the **IFB** and **IFNB** directives in macro definitions. You can use them to control conditional assembly of statements in the macro based on the parameters passed in the macro call. In such cases, *arg* should be one of the dummy parameters listed by the **MACRO** directive. For example:

```
IFB <>
```

This example tests the argument **<X>**. If this is in a macro definition and no parameter was passed for **X**, the directive would grant assembly.

The following example tests the argument **<&EXIT>**. This example is assumed to be in a macro definition. If no parameter is passed for **EXIT**, the directive does not grant assembly:

```
IFNB <&EXIT>
```

IFIDN and IFDIF Directives

Syntax

IFIDN <*arg1*>, <*arg2*>
IFDIF <*arg1*>, <*arg2*>

Description

The **IFIDN** and **IFDIF** directives test *arg1* and *arg2*. The **IFIDN** directive grants assembly if the arguments are identical. The **IFDIF** directive grants assembly if the arguments are different. The arguments can be any names, numbers, or expressions. To be identical, each character in *arg1* must match the corresponding character in *arg2*. You must use the angle brackets (< >).

Use the **IFIDN** and **IFDIF** directives in macro definitions. You can use them to control conditional assembly of statements in the macro based on the parameters passed in the macro call. In such cases, the arguments should be dummy parameters listed by the **MACRO** directive. For example:

```
IFIDN <X>, <Y>
```

This example tests the arguments **<X>** and **<Y>**. If this is in a macro definition and the parameters passed for **X** and **Y** are identical, the directive grants assembly.

```
IFDIF <&EXIT>, <CASE>
```

This example tests the arguments **<&EXIT>** and **<CASE>**. This is assumed to be in a macro definition. If the parameters passed for **EXIT** and **CASE** are identical, the directive does not grant assembly.

Chapter 8. Macro Directives

Introduction

This chapter explains how to create and use macros in your source files. The macro directives are:

MACRO
LOCAL
PURGE
REPT
IRP
IRPC
EXITM
ENDM

The **MACRO** directive lets you write a named block of source statements, then use that name in your source file to represent the statements. The assembler automatically replaces each occurrence of a macro name with the statements given in the macro definition. This means you can place a block of statements anywhere in your source file any number of times by simply defining it once, then giving the name where you need it. The LOCAL directive lets you define unique labels for a macro, and the **PURGE** directive lets you control the macros you define.

The **REPT**, **IRP**, and **IRPC** directives also let you create contiguous blocks of repeated statements. You control the number of times the statements are repeated. You can repeat them a given number of times, once for each parameter in a list, or once for each character in a string.

The macro directives use a special set of macro operators:

&	Ampersand
;;	Double semicolon
!	Exclamation mark
%	Percent sign

When used in a macro definition, these operators carry out special control operations, such as text substitution.

MACRO and ENDM Directives

Syntax

```
name    MACRO [dummy-parameter,,,]
        statements
                ENDM
```

Description

The **MACRO** and **ENDM** directives create a macro having the given *name* and containing the given *statements*.

The *name* must be a valid name and must be unique. It is used in the source file to invoke the macro. The *dummy-parameter* is a name that acts as a placeholder for values to be passed to the macro when it is called. You can give any number of dummy parameters, but they must all fit on one line. If you give more than one, you must separate them with commas (,). The *statements* are any valid assembler statements, including other **MACRO** directives. You can use any number of statements. You can use the dummy parameter any number of times in these statements.

A macro is called any time the macro's name appears in a source file. (The assembler ignores macro names in comments.) The assembler copies the statements in the macro definition to the point of call, replacing any dummy parameters in these statements with values passed in the call. Dummy parameters are replaced according to their order in the definition. Thus, the dummy parameter occupying the fourth parameter position in the macro definition is replaced by the fourth actual parameter in the macro call.

Macro definitions can be nested. This means a macro can be defined within another macro. The assembler does not process nested definitions until the outer macro has been called; therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to a depth limited only by the amount of memory available when the source file is assembled.

When a macro is redefined within itself, no instruction, directive, or comment can occur between the **ENDM** line of the redefinition and the **ENDM** of the enclosing macro body. Thus, the following is illegal:

```
xmpl        MACRO
              . . .
xmpl        MACRO
              . . .
            ENDM
; a comment which is illegal in this context
            ENDM
```

The corrected code follows:

```
xmpl        MACRO
              . . .
xmpl        MACRO
              . . .
            ENDM
            ENDM
```

Macro definitions can contain calls to other macros. These nested macro calls are expanded like any other macro call, but only when the outer macro is called. A macro definition cannot contain a call to itself.

The assembler replaces all occurrences of a dummy parameter's name, even if you do not intend it to. For example, if you use a register name such as **AX** or **BH** for a dummy, the assembler replaces all occurrences of that register name when it expands the macro. If the macro definition contains statements that use the register, not the dummy, the macro will be incorrectly expanded.

The assembler assembles the statements in the macro only if the macro is called, and only at the point the statements are inserted into the source file. This means all addresses in the assembled code are relative to the macro call, not the macro definition. The macro definition itself is never assembled.

You must be careful when using the word **MACRO** after the **TITLE**, **SUBTTL**, and **NAME** directives. The **MACRO** directive overrides these directives. Placing the word immediately after these directives causes the assembler to begin to create macros named **TITLE**, **SUBTTL**, and **NAME**. To avoid this problem, alter the word **MACRO** in some way when using it in a title or name. For example, prefix the word **MACRO** with a hyphen.

For example:

```
SUBTTL    addit -MACRO
addit     MACRO    xx,yy,zz
          mov      ax, xx
          add      ax, yy
          mov      zz, ax
          ENDM
```

This example defines a macro named `addit` that contains three statements and uses three dummy parameters. The dummy parameters are `xx`, `yy`, and `zz`. These parameters are replaced with actual values when the macro is called.

Macro Calls

Syntax

name [*actual-parameter*,,,]

Description

A ***macro call*** directs the assembler to copy the statements of the macro *name* to the point of call and to replace any dummy parameter in these statements with the corresponding actual parameter. The *name* must be the name of a macro defined earlier in the source file. The *actual-parameter* can be any name, number, or other value. You can give any number of actual parameters, but they must all fit on one line. Separate multiple parameters with commas, spaces, or tabs.

The assembler replaces the first dummy parameter with the first actual parameter, the second with the second, and so on. If a macro call has more actual parameters than dummy parameters, the extra actual parameters are ignored. If a call has fewer actual parameters, any remaining dummy parameters are replaced with nothing. This means that the assembler removes the dummy parameter name for the macro statements, but does nothing else.

If you wish to pass a list of values as a single actual parameter, you must place angle brackets (< >) around the list. You must separate the items in the list by commas.

For example:

```
ALLOCBLOCK 1,2,3,4,5
```

This example passes five numeric parameters to the macro `ALLOCBLOCK`.

```
ALLOCBLOCK <1,2,3,4,5>
```

This example passes one parameter to `ALLOCBLOCK`. This parameter is a list of five numbers.

```
add     1,inc,linecount
```

This example passes three parameters to the macro `add`. The first parameter is a number, but the second and third are symbols. The assembler replaces the corresponding dummy parameters with exactly what is typed here.

LOCAL Directive

Syntax

`LOCAL` *dummy-name*`,,,`

Description

The **LOCAL** directive creates unique names for use in macros. The *dummy-name* is a name for a placeholder that is to be replaced by the unique name when the macro is expanded. At least one *dummy-name* is required. If you give more than one, you must separate the names with commas. A dummy name can be used in any statement within the macro.

The assembler creates a new name for a dummy each time the macro is expanded. The name has the form

?? xxxx

where *xxxx* is a hexadecimal number in the range 0000 to FFFF.

You typically use the **LOCAL** directive to create unique labels for macros. Normally, if a macro containing a label is used more than once, the assembler displays a multiple-defined label error message because the same label will appear in both expansions. To avoid this problem, all labels can be local dummy names. The assembler guarantees that these names will be replaced with unique names whenever the macro is expanded, no matter how often the macro is expanded.

Note: You can use the directive only in a macro definition, and it must precede all other statements in the definition. For example:

```
loop    MACRO count, y
        LOCAL A
        mov   ax, y
A:      mov   cx, count
        inc   ax
        jnz   A
        ENDM
```

In this example, the **LOCAL** directive defines a dummy name **A** that is replaced with a unique name each time the macro is expanded. **A** is used in two places in the macro: as a statement label, and as the target of a **JNZ** instruction.

PURGE Directive

Syntax

`PURGE` *macro-name* `,,,`

Description

The **PURGE** directive deletes the current definition of the macro *macro-name* from memory. Any subsequent call to that macro causes the assembler to generate an error.

The **PURGE** directive is intended to clear memory space no longer needed by a macro. If the *macro-name* is an instruction or directive mnemonic, the directive restores its previous meaning.

The **PURGE** directive is often used with a ***macro library*** to let you choose from the library those macros you need in your source file. A macro library is simply a file containing macro definitions. You add this library to your source file using the **INCLUDE** directive, then remove unwanted definitions using the **PURGE** directive.

It is not necessary to **PURGE** a macro before redefining it. Any redefinition of a macro automatically purges the previous definition. Also, any macro can purge itself. For example:

```
PURGE add
```

This example deletes the macro named `add` from the assembler's memory.

```
PURGE mac1,mac2,mac9
```

This example deletes the macros named `mac1`, `mac2`, and `mac9`.

REPT and ENDM Directives

Syntax

`REPT` *expression*
    statements
`ENDM`

Description

The **REPT** and **ENDM** directives define a block of *statements* that are to be repeated *expression* number of times. The *expression* must evaluate to a 16-bit unsigned number. It must not contain external or undefined symbols. The *statements* can be any valid statements. For example:

```
X       =       0
        REPT    10
X       =       X+1
        DB      X
        ENDM
```

This example repeats the = and **DB** directives 10 times. The resulting statements create 10 bytes of data whose values range from 1 to 10.

IRP and ENDM Directives

Syntax

`IRP` *dummy*,*<parameter,,,>*
    statements
`ENDM`

Description

The **IRP** and **ENDM** directives define a block of *statements* that are to be repeated once for each *parameter* in the list enclosed by angle brackets (< >). The *dummy* is a name for a placeholder to be replaced by the current *parameter*. The *parameter* can be any legal symbol, string, numeric, or character constant. Any number of parameters can be given. If you give more than one, you must separate them with commas. The *statements* can be any valid assembler statements. The dummy can be used any number of times in these statements.

When the assembler encounters an **IRP** directive, it makes one copy of the statements for each parameter in the enclosed list. While copying the statements, it replaces all occurrences of dummy in these statements with the current parameter. If a null parameter (< >) is found in the list, the dummy is replaced with a null value. If the parameter list is empty, the **IRP** directive is ignored and no statements are copied. For example:

```
IRP  X,<1,2,3,4,5,6,7,8,9,10>
DB   X
ENDM
```

This example repeats the **DB** directive 10 times, once for each number in the list. The resulting statements create 10 bytes of data having the values 1 through 10.

Note: If an empty parameter is found in the list, the assembler repeats the statements once, replacing the dummy with no value. If the entire list is empty, the assembler skips the repeat block.

If you use an **IRP** directive inside a macro definition and the parameter list of the **IRP** directive is also a dummy parameter of the macro, you must enclose that dummy parameter within angle brackets. For example, in the following macro definition, the dummy parameter **X** is used as the parameter list for the **IRP** directive:

```
alloc   MACRO        X
            IRP      Y,<X>
            DB       Y
            ENDM
        ENDM
```

If this macro is called with

```
alloc <1,2,3,4,5,6,7,8,9,10>
```

the macro expansion becomes

```
IRP     Y,<1,2,3,4,5,6,7,8,9,10>
DB      Y
ENDM
```

That is, the macro removes the brackets from the actual parameter before replacing the dummy. This means you must provide the angle brackets for the parameter list yourself.

IRPC and ENDM Directives

Syntax

```
IRPC dummy,string
     statements
ENDM
```

Description

The **IRPC** and **ENDM** directives define a block of *statements* that is repeated once for each character in the *string*. The *dummy* is a name for a placeholder to be replaced by the current character in the *string*. The *string* can be any combination of letters, digits, and other characters. The string should be enclosed with angle brackets (< >) if it contains spaces, commas, or other separating characters. The *statements* can be any valid assembler statements. You can use the dummy any number of times in these statements.

When the assembler encounters an **IRPC** directive, it makes one copy of the statements for each character in the string. While copying the statements, it replaces all occurrences of dummy in these statements with the current character. For example:

```
IRPC  X,0123456789
DB    X+1
ENDM
```

This example repeats the **DB** directive 10 times, once for each character in the string `0123456789`. The resulting statements create 10 bytes of data having the values 1 through 10.

EXITM Directive

Syntax

```
EXITM
```

Description

The **EXITM** directive directs the assembler to terminate macro or repeat block expansion and continue assembly with the next statement after the macro call or repeat block. You typically use the directive with **IF** directives to allow conditional expansion of the last statements in a macro or repeat block.

When the assembler encounters an **EXITM**, it exits the macro or repeat block immediately. It does not process any remaining statements. If the assembler encounters **EXITM** in a macro or repeat block nested in another, it returns to expanding the outer level block. For example:

```
alloc   MACRO   times
        x  =    0
                REPT    times
                IFE     x-0FFH
                        EXITM
                ELSE
                        DB      x
                ENDIF
                x  =    x+1
                ENDM
        ENDM
```

This example defines a macro that creates no more than 255 bytes of data. The macro contains an **IFE** directive that checks the expression `x-0FFH`. When this expression is 0 (`x` equal to 255), the **EXITM** directive is processed and expansion of the macro stops.

Substitute Operator

Syntax

`&`*dummy-parameter*
`or`
dummy-parameter`&`

Description

The ***substitute operator*** (**&**) forces the assembler to replace the given *dummy-parameter* with its corresponding actual parameter value. Use this operator anywhere a dummy parameter immediately precedes or follows other characters, or any time the parameter appears in a quoted string. For example:

```
errgen          MACRO  Y,X
error&X              db        'Error &Y - &X'
                ENDM
```

In this example, the assembler replaces `&X` with the value of the actual parameter passed to the macro `errgen`. If you call the macro with the statement:

```
errgen 1,wait
```

the macro expands to:

```
errorwait       db      'Error 1 - wait'
```

Note: For complex, nested macros, you can use extra ampersands to delay the actual replacement of a dummy parameter. In general, you need to supply as many ampersands as there are levels of nesting.

For example, in the following macro definition, the substitute operator is used twice with `Z` to make sure its replacement occurs while the **IRP** directive is being processed.

```
alloc   MACRO  X
             IRP      Z,<1,2,3>
        X&&Z DB       Z
             ENDM
        ENDM
```

In this example, the dummy parameter `X` is replaced immediately when the macro is called. The dummy parameter `Z`, however, is not replaced until the **IRP** directive is processed. This means the parameter is replaced once for each number in the **IRP** parameter list. If the macro is called with:

```
alloc VAR
```

the expanded macro will be:

```
VAR1  DB  1
VAR2  DB  2
VAR3  DB  3
```

Literal Text Operator

Syntax

<*text*>

Description

The ***literal text operator*** directs the assembler to treat *text* as a single literal. The operator is most often used with macro calls and the **IRP** directive to ensure that values in a parameter list are treated as a single parameter.

You also can use the literal text operator to force the assembler to treat special characters such as **;** or **&** literally. For example, the semicolon inside angle brackets `<;>` becomes a semicolon, not a comment indicator.

The assembler removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting.

Literal Character Operator

Syntax

`!`*character*

Description

The ***literal character operator*** forces the assembler to treat *character* as a literal. For example, you can force the assembler to treat special characters such as ; or & literally. Therefore, !; is equivalent to `<;>`.

Expression Operator

Syntax

`%`*text*

Description

The ***expression operator*** (%) causes the assembler to treat *text* as an expression. The assembler computes the expression's value, using numbers of the current radix, and replaces *text* with this new value. The *text* must represent a valid assembler expression.

You typically use the expression operator in macro calls where you need to pass the result of an expression to the macro instead of the actual expression. For example:

```
print  MACRO   msg,n
        %OUT  * msg,n *
        ENDM
sym1    EQU  100
sym2    EQU  200
        print <sym1 + sym2 = >,%(sym1 + sym2)
```

In this example, the macro call:

```
print  <sym1 + sym2 = >,%(sym1 + sym2)
```

passes the text literal `sym1 + sym2 =` to the dummy parameter `msg`. It passes the value 300, the result of the expression `(sym1 + sym2)`, to the dummy `n`.

Macro Comment

Syntax

`;;`*text*

Description

The ***macro comment*** is any text in a macro definition that does not need to be copied in the macro expansion. The assembler ignores all text following the double semicolon (;;) and the text appears only in the macro definition when the source listing is created.

Regular comments, unlike macro comments, can be copied to each macro expansion by specifying the **.XALL** directive in the source file.

Chapter 9. File Control

Introduction

This chapter describes ***file control directives***. These directives provide control of the source, object, and listing files read and created by the Macro Assembler during an assembly.

The file control directives are:

INCLUDE	Include a source file
.RADIX	Alter default input radix
%OUT	Display message on console
NAME	Copy name to object file
TITLE	Set program listing title
SUBTTL	Set program listing subtitle
PAGE	Set program listing page size
.LIST	List statements in program listing
.XLIST	Suppress listing statements
.SFCOND	Suppress false conditional listing
.LFCOND	List false conditional in program listing
.TFCOND	Toggle false conditional listing
.LALL	List macro expansions in program listing
.XALL	Exclude comments from macro listing
.SALL	Suppress listing macro expansion
.CREF	List symbols in cross-reference file (not used for PC XENIX)
.XCREF	Suppress symbol listing (not used for PC XENIX).

The following sections describe the directives in detail.

INCLUDE Directive

Syntax

`INCLUDE` *filename*

Description

The **INCLUDE** directive inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must name an existing file. You must give a path name if the file is not in the current working directory. If it does not find the named file, the assembler displays an error message and stops.

When the assembler encounters an **INCLUDE** directive, it opens the named file and begins to assemble its source statements immediately. When finished reading all statements, the assembler resumes with the next statement following the directive.

You can have nested **INCLUDE** directives. This means a file named by an **INCLUDE** directive can contain its own **INCLUDE** directives.

When it creates a program listing, the assembler marks included statements with the letter C.

The following examples are **INCLUDE** directives:

```
include entry
include include/record
include /usr/include/as/stdio
```

.RADIX Directive

Syntax

`.RADIX` *expression*

Description

The **.RADIX** directive sets the default input radix for numbers in the source file. The *expression* defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base. It must be within the range 2 to 16. The following lists some common values:

```
2       - binary
8       - octal
10      - decimal
16      - hexadecimal
```

The *expression* is always considered a decimal number regardless of the current default radix. For example:

```
.radix  16
.radix  2
```

Note: The **.RADIX** directive does not affect the **DD**, **DQ**, or **DT** directives. Numbers entered in the expression of these directives are always evaluated as decimal unless a numeric suffix is appended to the value.

The **.RADIX** directive does not affect the optional radix specifiers, **B** and **D**, used with integers. When **B** or **D** appears at the end of any integer, it always is considered to be a radix specifier even if the current input radix is 16. This means that numbers such as `0abcd` and `234b` are illegal even when the input radix is set to 16. (`a`, `b`, and `c` are not legal digits for a decimal number. Similarly, `2`, `3`, and `4` are not legal for a binary number.)

%OUT Directive

Syntax

`%OUT` *text*

Description

The **%OUT** directive directs the assembler to display the *text* at the user's terminal. The directive is useful for displaying messages during specific points of a long assembly.

The **%OUT** directive generates output for both assembly passes. You can use the **IF1** and **IF2** directives to control when the assembler processes the directive. For example:

```
IF1
        %OUT First Pass -- Okay
ENDIF
```

NAME Directive

Syntax

NAME *module-name*

Description

The **NAME** directive sets the name of the current module to *module-name*. The linker uses a module name when displaying error messages.

The *module-name* can be any combination of letters and digits. Although the name can be any length, the assembler only uses the first 6 characters. The name must be unique and not be a reserved word. For example:

```
name main
```

If you do not use the NAME directive, the assembler creates a default module name using the first 6 characters of a **TITLE** directive. If the assembler does not find a **TITLE** directive, it uses the default name **A**.

TITLE Directive

Syntax

TITLE *text*

Description

The **TITLE** directive defines the program listing title. It directs the assembler to copy *text* to the first line of each new page in the program listing. The *text* can be any combination of characters up to 60 characters in length.

The assembler does not allow more than one **TITLE** directive per module. For example:

```
title PROG1 -- 1st Program
```

The assembler uses the first 6 nonblank characters of the title as the module name if the module does not contain a **NAME** directive.

SUBTITLE Directive

Syntax

`SUBTTL` *text*

Description

The **SUBTTL** directive defines the listing subtitle. It directs the assembler to copy *text* to the line immediately after the title on each new page in the program listing. The *text* can be any combination of characters. The assembler uses only the first 60 characters. If you do not give characters, the subtitle line is left blank.

You can give any number of **SUBTTL** directives in a program. Each new directive replaces the current subtitle with the new *text*. For example:

```
subttl   SPECIAL I/O ROUTINE
```

This example creates the subtitle **SPECIAL I/O ROUTINE**.

```
subttl
```

This example creates a blank subtitle.

PAGE Directive

Syntax

`PAGE` *length*, *width*
`PAGE +`
`PAGE`

Description

The **PAGE** directive sets the line length and character width of the program listing, increments section page numbering, or generates a page break in the listing.

If you give a *length* and a *width*, **PAGE** sets the maximum number of lines per page to *length*, and the maximum number of characters per line to *width*. The *length* must be in the range 10 to 255. The default is 50. The *width* must be in the range 60 to 132. The default is 80. You can give a *width* without a *length* as long as the comma (,) precedes the *width*.

If you give a plus sign (+.), **PAGE** increments the section number and resets the page number to 1. Program listing page numbers have the form:

section - minor

By default, page numbers start at 1-1.

If you do not give an argument, **PAGE** starts a new output page in the program listing. It copies a form feed character to the file and generates a title and subtitle line. For example:

```
PAGE
```

This example creates a page break.

```
PAGE 58,60
```

This example sets the maximum page length to 58 lines, and the maximum width to 60 characters.

For example:

```
PAGE ,132
```

This example sets the maximum width to 132 characters. The current page length remains unchanged.

```
PAGE +
```

This example increments the current section number and sets the page number to 1.

.LIST and .XLIST Directives

Syntax

```
.LIST
.XLIST
```

Description

The **.LIST** and **.XLIST** directives control which source program lines are copied to the program listing. The **.XLIST** directive suppresses copying of subsequent source lines to the program listing. The **.LIST** directive restores copying. You typically use the directives in pairs to prevent the assembler from copying a section of a given source file to the program listing.

The **.XLIST** directive overrides all other listing directives. For example:

```
.XLIST
        ;listing suspended here
.LIST
        ;listing resumes here
```

.SFCOND, .LFCOND, and .TFCOND Directives

Syntax

```
.SFCOND
.LFCOND
.TFCOND
```

Description

The **.SFCOND** and **.LFCOND** directives determine whether the assembler should list conditional blocks. The **.SFCOND** directive suppresses the listing of any subsequent conditional blocks whose **IF** condition is false.

The **.LFCOND** directive restores the listing of these blocks. You can use these directives like **.LIST** and **.XLIST** to suppress listing of the conditional blocks in sections of a program.

The **.TFCOND** directive sets the default mode for listing of conditional blocks. This directive works with the **-X** option of the assembler. If **-X** is not entered in the **as** command line, **.TFCOND** causes the assembler to list false conditional blocks by default. If **-X** is entered, **.TFCOND** causes the assembler to suppress false conditional blocks. For example:

```
.SFCOND
IF 0
        ;This block will not be listed.
ENDIF
.LFCOND
IF 0
        ;This block will be listed.
ENDIF
```

.LALL, .XALL, and .SALL Directives

Syntax

```
.LALL
.XALL
.SALL
```

Description

The **.LALL**, **.XALL**, and **.SALL** directives control the listing of the statements in macros that have been expanded in the source file. The assembler always lists the full macro definition, but lists macro expansions only if the appropriate directive is set.

The **.LALL** directive causes the assembler to list all the source statements in a macro, including comments preceded by a single semicolon (;) but not those preceded by a double semicolon (;;).

The **.XALL** directive lists only those source statements that generate code or data, so comments are ignored.

The **.SALL** directive suppresses listing of all macro expansions. That is, the assembler copies the macro call to the source listing, but does not copy the source lines that the call generates.

.XALL is in effect when the assembler first begins execution. For example:

```
.SALL
        ;No macros listed here.
.LALL
        ;Macros listed in full.
.XALL
        ;Macros listed by generated code or data only.
```

.CREF and .XCREF Directives

Syntax

.CREF
.XCREF *name*,,,

Description

The **.CREF** and **.XCREF** directives control the generation of cross-references for the macro assembler's cross-reference file for DOS transfer usage only. The **.XCREF** directive suppresses the generation of label, variable, and symbol cross-references. The **.CREF** function restores this generation.

If you give a *name* with **.XCREF**, the assembler suppresses only that label, variable, or symbol. The assembler cross-references all other names.

The named label, variable, or symbol also is omitted from the symbol table of the program listing. If you give two or more names, separate them with commas. For example:

```
.XCREF one, two, three
```

PC XENIX Cross-Referencing

For PC XENIX, a cross-reference can be obtained using the PC XENIX **xref** command. For example:

`xref` *file.s*

This example directs **xref** to output a cross-reference for the program named *file.s* to the standard output. For more information on **xref** see *XENIX Commands Reference.*

Appendix A. Instruction Summary

The Macro Assembler can assemble instructions for the 8086, 80186, and 80286 microprocessors and the 8087 and 80287 floating-point coprocessors. The assembler assembles any program you write for an 8086, 80186, or 80286 microprocessor environment as long as the program uses the instruction syntax described in this appendix.

By default, the assembler recognizes 8086, 8087, 80286, and 80287 nonprotected instructions only. If a source program contains 80186 or 80287 protected instructions, you must use one or more instruction set directives in the source file to enable assembly of the instructions. The following sections list the syntax of all the instructions and the instruction set directives that the assembler recognizes.

Here are the abbreviations you can use in the syntax descriptions:

Symbol	**Meaning**
accum	Accumulator: AX or AL
reg	Byte or word register Byte: AL, AH, BL, BH, CL, CH, DL, DH Word: AX, BX, CX, DX, SI, DI, BP, SP
segreg	Segment register: CS, DS, SS, ES
r/m	General operand: register, memory address, indexed operand, based operand, or based indexed operand
immed	8- or 16-bit immediate value: constant or symbol
mem	Memory operand: label, variable, or symbol
label	Instruction label.

8086 Instructions

The following is a complete list of the 8086 instructions. The assembler assembles all 8086 instructions by default.

Syntax	**Action**
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC *accum, immed*	Add immediate with carry to accumulator
ADC *r/m, immed*	Add immediate with carry to operand
ADC *r/m, reg*	Add register with carry to operand
ADC *reg, r/m*	Add operand with carry to register
ADD *accum, immed*	Add immediate to accumulator
ADD *r/m, immed*	Add immediate to operand
ADD *r/m, reg*	Add register to operand
ADD *reg, r/m*	Add operand to register
AND *accum, immed*	Bitwise AND immediate with accumulator
AND *r/m, immed*	Bitwise AND immediate with operand
AND *r/m, reg*	Bitwise AND register with operand
AND *reg, r/m*	Bitwise AND operand with register
CALL *label*	Call instruction at label
CALL *r/m*	Call instruction indirect
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP *accum, immed*	Compare immediate with accumulator

CMP *r/m, immed*	Compare immediate with operand
CMP *r/m, reg*	Compare register with operand
CMP *reg, r/m*	Compare operand with register
CMPS *src, dest*	Compare strings
CMPSB	Compare strings byte for byte
CMPSW	Compare strings word for word
CWD	Convert word to double word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC *r/m*	Decrement operand
DEC *reg*	Decrement 16-bit register
DIV *r/m*	Divide accumulator by operand
ESC *immed, r/m*	Escape with 6-bit immediate and operand
HLT	Halt
IDIV *r/m*	Integer divide accumulator by operand
IMUL *r/m*	Integer multiply accumulator by operand
IN *accum, immed*	Input from port (8-bit immediate)
IN *accum*, DX	Input from port given by DX
INC *r/m*	Increment operand
INC *reg*	Increment 16-bit register
INT 3	Software interrupt 3 (encoded as one byte)
INT *immed*	Software interrupt 0 through 255
INTO	Interrupt on overflow
IRET	Return from interrupt
JA *label*	Jump on above
JAE *label*	Jump on above or equal
JB *label*	Jump on below
JBE *label*	Jump on below or equal
JC *label*	Jump on carry

JCXZ *label*	Jump on CX zero
JE *label*	Jump on equal
JG *label*	Jump on greater
JGE *label*	Jump on greater or equal
JL *label*	Jump on less than
JLE *label*	Jump on less than or equal
JMP *label*	Jump to instruction at label
JMP *r/m*	Jump to instruction indirect
JNA *label*	Jump on not above
JNAE *label*	Jump on not above or equal
JNB *label*	Jump on not below
JNBE *label*	Jump on not below or equal
JNC *label*	Jump on no carry
JNE *label*	Jump on not equal
JNG *label*	Jump on not greater
JNGE *label*	Jump on not greater or equal
JNL *label*	Jump on not less than
JNLE *label*	Jump on not less than or equal
JNO *label*	Jump on not overflow
JNP *label*	Jump on not parity
JNS *label*	Jump on not sign
JNZ *label*	Jump on not zero
JO *label*	Jump on overflow
JP *label*	Jump on parity
JPE *label*	Jump on parity even
JPO *label*	Jump on parity odd
JS *label*	Jump on sign
JZ *label*	Jump on zero
LAHF	Load AH with flags
LDS *r/m*	Load operand into DS

LEA *r/m*	Load effective address of operand
LES *r/m*	Load operand into ES
LOCK	Lock bus
LODS *src*	Load string
LODSB	Load byte from string into AL
LODSW	Load word from string into AX
LOOP *label*	Loop
LOOPE *label*	Loop while equal
LOOPNE *label*	Loop while not equal
LOOPNZ *label*	Loop while not zero
LOOPZ *label*	Loop while zero
MOV *accum, mem*	Move memory to accumulator
MOV *mem, accum*	Move accumulator to memory
MOV *r/m, immed*	Move immediate to operand
MOV *r/m, reg*	Move register to operand
MOV *r/m, segreg*	Move segment register to operand
MOV *reg, immed*	Move immediate to register
MOV *reg, r/m*	Move operand to register
MOV *segreg, r/m*	Move operand to segment register
MOVS *dest, src*	Move string
MOVSB	Move string byte by byte
MOVSW	Move string word by word
MUL *r/m*	Multiply accumulator by operand
NEG *r/m*	Negate operand
NOP	No operation
NOT *r/m*	Invert operand bits
OR *accum, immed*	Bitwise OR immediate with accumulator
OR *r/m, immed*	Bitwise OR immediate with operand
OR *r/m, reg*	Bitwise OR register with operand
OR *reg, r/m*	Bitwise OR operand with register

OUT DX,*accum*	Output to port given by DX
OUT *immed, accum*	Output to port (8-bit immediate)
POP *r/m*	Pop 16-bit operand
POP *reg*	Pop 16-bit register from stack
POP *segreg*	Pop segment register
POPF	Pop flags
PUSH *r/m*	Push 16-bit operand
PUSH *reg*	Push 16-bit register onto stack
PUSH *segreg*	Push segment register
PUSHF	Push flags
RCL *r/m*, 1	Rotate left through carry by 1 bit
RCL *r/m*, CL	Rotate left through carry by CL
RCR *r/m*, 1	Rotate right through carry by 1 bit
RCR *r/m*, CL	Rotate right through carry by CL
REP	Repeat string operation
REPE	Repeat if equal
REPNE	Repeat if not equal
REPNZ	Repeat if not zero
REPZ	Repeat if zero
RET [*immed*]	Return after popping bytes from stack
ROL *r/m*, 1	Rotate left by 1 bit
ROL *r/m*, CL	Rotate left by CL
ROR *r/m*, 1	Rotate right by 1 bit
ROR *r/m*, CL	Rotate right by CL
SAHF	Store AH into flags
SAL *r/m*, 1	Shift arithmetic left by 1 bit
SAL *r/m*, CL	Shift arithmetic left by CL
SAR *r/m*, 1	Shift arithmetic right by 1 bit
SAR *r/m*, CL	Shift arithmetic right by CL

SBB *accum, immed*	Subtract immediate from accumulator and carry flag
SBB *r/m, immed*	Subtract immediate from operand and carry flag
SBB *r/m, reg*	Subtract register and carry flag
SBB *reg, r/m*	Subtract operand and carry flag
SCAS *dest*	Scan string
SCASB	Scan string for byte in AL
SCASW	Scan string for word in AX
SHL *r/m*, 1	Shift left by 1 bit
SHL *r/m*, CL	Shift left by CL
SHR *r/m*, 1	Shift right by 1 bit
SHR *r/m*, CL	Shift right by CL
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS *dest*	Store string
STOSB	Store byte in AL at string
STOSW	Store word in AX at string
SUB *accum, immed*	Subtract immediate from accumulator
SUB *r/m, immed*	Subtract immediate from operand
SUB *r/m, reg*	Subtract register from operand
SUB *reg, r/m*	Subtract operand from register
TEST *accum, immed*	Compare immediate bits with accumulator
TEST *r/m, immed*	Compare immediate bits with operand
TEST *r/m, reg*	Compare register bits with operand
TEST *reg, r/m*	Compare operand bits with register
WAIT	Wait
XCHG *accum, reg*	Exchange accumulator with register
XCHG *r/m, reg*	Exchange operand with register

XCHG *reg, accum*	Exchange register with accumulator
XCHG *reg, r/m*	Exchange register with operand
XLAT *mem*	Translate
XOR *accum, immed*	Bitwise XOR immediate with accumulator
XOR *r/m, immed*	Bitwise XOR immediate with operand
XOR *r/m, reg*	Bitwise XOR register with operand
XOR *reg, r/m*	Bitwise XOR operand with register.

The string instructions (**CMPS**, **LODS**, **MOVS**, **SCAS**, and **STOS**) use the **DS**, **SI**, **ES**, and **DI** registers to compute operand locations. Source operands are assumed to be at **DS:fflSI“**; destination operands at **ES:fflDI“**.

The operand type (**BYTE** or **WORD**) is defined by the instruction mnemonic. For example, **CMPSB** specifies **BYTE** operands and **CMPSW** specifies **WORD** operands. For the **CMPS**, **LODS**, **MOVS**, **SCAS**, and **STOS** instructions, the *src* and *dest* operands are dummy operands that define the operand type only. The offsets associated with these operands are not used. The *src* operand can also be used to specify a segment override. The **ES** register for the destination operand cannot be overridden. For example:

```
cmps  word ptr string, word ptr es:0
lods  byte ptr string
movs  byte ptr es:0,  byte ptr string
```

The **REP**, **REPE**, **REPNE**, **REPNZ**, or **REPZ** instructions provide a way to repeatedly execute a string instruction for a given count or while a given condition is true. If a repeat instruction immediately precedes a string instruction (both instructions must be on the same line), the instructions are repeated until the specified repeat condition is false, or the **CX** register is equal to zero. The repeat instruction decrements **CX** by one for each execution. For example:

```
mov     cx, 10
rep     scasb
```

In this example, **SCASB** is repeated 10 times.

8087 Instructions

The following is a list of the 8087 instructions. The assembler assembles all 8087 instructions by default.

Syntax	**Action**
F2XM1	Calculate 2^x-1
FABS	Take absolute value of top of stack
FADD	Add real
FADD *mem*	Add real from memory
FADD ST, ST(*i*)	Add real from stack
FADD ST (*i*), ST	Add real to stack
FADDP ST (*i*), ST	Add real and pop stack
FBLD *mem*	Load 10-byte packed decimal on stack
FBSTP *mem*	Store 10-byte packed decimal and pop
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after WAIT
FCOM	Compare real
FCOM ST	Compare real with top of stack
FCOM ST(*i*)	Compare real with stack
FCOMP	Compare real and pop stack
FCOMP ST	Compare real with top of stack and pop
FCOMP ST(*i*)	Compare real with stack and pop stack
FCOMPP	Compare real and pop stack twice
FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real
FDIV *mem*	Divide real from memory
FDIV ST, ST(*i*)	Divide real from stack
FDIV ST(*i*), ST	Divide real in stack
FDIVP ST(*i*), ST	Divide real and pop stack

FDIVR	Reverse real divide
FDIVR *mem*	Reverse real divide from memory
FDIVR ST, ST(*i*)	Reverse real divide from stack
FDIVR ST(*i*), ST	Reverse real divide in stack
FDIVRP ST(*i*), ST	Reverse real divide and pop stack twice
FENI	Enable interrupts after WAIT
FFREE	Free stack element
FFREE ST	Free top of stack element
FFREE ST(*i*)	Free *i*th stack element
FIADD *mem*	Add 2- or 4-byte integer
FICOM *mem*	2- or 4-byte integer compare
FICOMP *mem*	2- or 4-byte integer compare and pop stack
FIDIV *mem*	2- or 4-byte integer divide
FIDIVR *mem*	Reverse 2- or 4-byte integer divide
FILD *mem*	Load 2-, 4-, or 8-byte integer on stack
FIMUL *mem*	2- or 4-byte integer multiply
FINCSTP	Increment stack pointer
FINIT	Initialize processor after WAIT
FIST *mem*	Store 2- or 4-byte integer
FISTP *mem*	Store 2-, 4-, or 8-byte integer and pop stack
FISUB *mem*	2- or 4-byte integer subtract
FISUBR *mem*	Reverse 2- or 4-byte integer subtract
FLD *mem*	Load 4-, 8-, or 10-byte real on stack
FLD1	Load +1.0 onto top of stack
FLDCW *mem*	Load control word
FLDENV *mem*	Load 8087 environment (14-bytes)
FLDL2E	Load $\log_2 e$ onto top of stack
FLDL2T	Load $\log_2 10$ onto top of stack
FLDLG2	Load $\log_{10} 2$ onto top of stack

FLDLN2	Load $\log_e 2$ onto top of stack
FLDPI	Load pi onto top of stack
FLDZ	Load +0.0 onto top of stack
FMUL	Multiply real
MUL *mem*	Multiply real from memory
FMUL ST, ST(*i*)	Multiply real from stack
FMUL ST(*i*), ST	Multiply real to stack
FMULP ST(*i*), ST	Multiply real and pop stack
FNCLEX	Clear exceptions with no WAIT
FNDISI	Disable interrupts with no WAIT
FNENI	Enable interrupts with no WAIT
FNINIT	Initialize processor, with no WAIT
FNOP	No operation
FNSAVE *mem*	Save 8087 state (94 bytes) with no WAIT
FNSTCW *mem*	Store control word with no WAIT
FNSTENV *mem*	Store 8087 environment with no WAIT
FNSTSW *mem*	Store 8087 status word with no WAIT
FPATAN	Partial arctangent function
FPREM	Partial remainder
FPTAN	Partial tangent function
FRNDINT	Round to integer
FRSTOR *mem*	Restore 8087 state (94 bytes)
FSAVE *mem*	Save 8087 state (94 bytes) after WAIT
FSCALE	Scale
FSQRT	Square root
FST	Store real
FST ST	Store real from top of stack
FST ST(*i*)	Store real from stack
FSTCW *mem*	Store control word with WAIT
FSTENV *mem*	Store 8087 environment after WAIT

FSTP *mem*	Store 4-, 8-, or 10-byte real and pop stack
FSTSW *mem*	Store 8087 status word after WAIT
FSUB	Subtract real
FSUB *mem*	Subtract real from memory
FSUB ST, ST(*i*)	Subtract real from stack
FSUB ST(*i*), ST	Subtract real to stack
FSUBP ST(*i*), ST	Subtract real and pop stack
FSUBR	Reverse real subtract
FSUBR *mem*	Reverse real subtract from memory
FSUBR ST, ST(*i*)	Reverse real subtract from stack
FSUBR ST(*i*), ST	Reverse real subtract in stack
FSUBRP ST(*i*), ST	Reverse real subtract and pop stack
FTST	Test top of stack
FWAIT	Wait for last 8087 operation to complete
FXAM	Examine top of stack element
FXCH	Exchange contents of stack elements
FFREE ST	Exchange top of stack element
FFREE ST(*i*)	Exchange top of stack and *i*th element
FXTRACT	Extract exponent and significand
FYL2X	Calculate Y $\log_2 x$
FYL2PI	Calculate Y $\log_2(x+1)$.

80186 Instructions

The 80186 instruction set consists of all 8086 instructions plus the following instructions. You can use the **.186** directive to enable these instructions for assembly.

Syntax	**Action**
BOUND *reg, mem*	Detect value out of range
ENTER *immed16, immed8*	Enter procedure
IMUL *reg, immed*	Integer multiply register by immediate
IMUL *reg, r/m, immed*	Integer multiply operand by immediate and save in reg
INS *mem*, DX	Input string from port DX
INSB *mem*, DX	Input byte string from port DX
INSW *mem*, DX	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, *mem*	Output byte or word or string to port DX
OUTSB DX, *mem*	Output byte string to port DX
OUTSW DX, *mem*	Output word string to port DX
PUSH *immed*	Push immediate
PUSHA	Push all registers
POPA	Pop all registers
RCL *r/m, immed*	Rotate left through carry by immediate
RCR *r/m, immed*	Rotate right through carry by immediate
ROL *r/m, immed*	Rotate left by immediate
ROR *r/m, immed*	Rotate right by immediate
SAL *r/m, immed*	Shift arithmetic left by immediate

SAR *r/m, immed*	Shift arithmetic right by immediate
SHL *r/m, immed*	Shift left by immediate
SHR *r/m, immed*	Shift right by immediate.

80286 Nonprotected Instructions

The 80286 nonprotected instruction set consists of all 8086 instructions plus the following instructions. You can use the **.286c** directive to enable these instructions for assembly.

Syntax	**Action**
BOUND *reg, mem*	Detect value out of range
ENTER *immed16, immed8*	Enter procedure
IMUL *reg, immed*	Integer multiply register by immediate
IMUL *reg, r/m, immed*	Integer multiply operand by immediate and save in reg
INS *mem*, DX	Input string from port DX
INSB *mem*, DX	Input byte string from port DX
INSW *mem*, DX	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, *mem*	Output byte or word or string to port DX
OUTSB DX, *mem*	Output byte string to port DX
OUTSW DX, *mem*	Output word string to port DX
PUSH *immed*	Push immediate
PUSHA	Push all registers
POPA	Pop all registers
RCL *r/m, immed*	Rotate left through carry by immediate
RCR *r/m, immed*	Rotate right through carry by immediate
ROL *r/m, immed*	Rotate left by immediate

ROR *r/m, immed*	Rotate right by immediate
SAL *r/m, immed*	Shift arithmetic left by immediate
SAR *r/m, immed*	Shift arithmetic right by immediate
SHL *r/m, immed*	Shift left by immediate
SHR *r/m, immed*	Shift right by immediate.

80286 Protected Instructions

The 80286 protected instruction set consists of all 8086 and 80286 nonprotected instructions plus the following instructions. You can use the **.286p** directive to enable these instructions for assembly.

Syntax	**Action**
ARPL *mem, reg*	Adjust requested privilege level
CLTS	Clear task switched flag
LAR *reg, mem*	Load access rights
LGDT *mem*	Load global descriptor table (8 bytes)
LIDT *mem*	Load interrupt descriptor table (8 bytes)
LLDT *mem*	Load local descriptor table
MSW *mem*	Load machine status word
LSL *reg, mem*	Load segment limit
LTR *mem*	Load task register
SGDT *mem*	Store global descriptor table (8 bytes)
SIDT *mem*	Store interrupt descriptor table (8 bytes)
SLDT *mem*	Store local descriptor table
SMSW *mem*	Store machine status word
STR *mem*	Store task register
VERR *mem*	Verify read access
VERW *mem*	Verify write access.

80287 Instructions

The 80287 instruction set consists of all 8087 instructions plus the following additional instructions. You can use the **.287** directive to enable these instructions for assembly.

Syntax	Action
FSETPM	Set Protected Mode
FSTSW AX	Store Status Word in AX (WAIT)
FNSTSW AX	Store Status Word in AX (no WAIT).

Appendix B. Directive Summary

Introduction

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions. The directives are:

.186
.286c
.286p
.287
.8086
.8087
=
ASSUME
COMMENT
.CREF
DB
DD
DQ
DT
DW
ELSE
END
ENDIF
ENDP
ENDS
EQU
EVEN
EXTRN
GROUP
IF
IF1
IF2
IFB
IFDEF
IFDIF
IFE
IFIDN
IFNB
IFNDEF
INCLUDE
LABEL
.LALL
.LFCOND
.LIST
NAME
ORG
%OUT
PAGE
PROC
PUBLIC
.RADIX
RECORD
.SALL
SEGMENT
.SFCOND
STRUC
SUBTTL
.TFCOND
TITLE
.XALL
.XCREF
.XLIST

You can use any combination of uppercase and lowercase letters when giving directive names in a source file.

Directive Syntax and Function

The following is a complete list of directive syntax and function.

.186

Explanation: Enables assembly of 80186 instructions.

.286c

Explanation: Enables assembly of 80286 unprotected instructions.

.286p

Explanation: Enables assembly of 80286 protected instructions.

.287

Explanation: Enables assembly of 80287 instructions.

.8086

Explanation: Enables assembly of 8086 instructions while disabling assembly of 80186 and 80286 instructions.

.8087

Explanation: Enables assembly of 8087 instructions while disabling assembly of 80287 instructions.

name* = *expression

Explanation: Assigns the numeric value of *expression* to *name.*

ASSUME *seg-reg* : *seg-name* ,,,

Explanation: Selects the given segment register *seg-reg* to be the default segment register for all symbols in the named segment or group. If *seg-name* is NOTHING, no register is selected.

COMMENT *delim text delim*

Explanation: Treats all *text* between the given pair of delimiters *delim* as a comment.

.CREF

Explanation: Restores listing of symbols in the cross-reference listing file (for DOS transfer usage only).

[*name*] DB *initial-value* ,,,

Explanation: Allocates and initializes a byte (8 bits) of storage for each *initial-value.*

[*name*] DW *initial-value* ,,,

Explanation: Allocates and initializes a word (2 bytes) of storage for each given *initial-value.*

[*name*] DD *initial-value* ,,,

Explanation: Allocates and initializes a doubleword (4 bytes) of storage for each given *initial-value.*

[*name*] DQ *initial-value* ,,,

Explanation: Allocates and initializes a quadword (8 bytes) of storage for each given *initial-value.*

[*name*] DT *initial-value* ,,,

Explanation: Allocates and initializes 10 bytes of storage for each given *initial-value.*

ELSE

Explanation: Marks the beginning of an alternate block within a conditional block.

END [*expression*]

Explanation: Marks the end of the module and optionally sets the program entry point to *expression.*

ENDIF

Explanation: Terminates a conditional block.

name **ENDP**

Explanation: Marks the end of a procedure definition.

name **ENDS**

Explanation: Marks the end of a segment or structure type definition.

name **EQU** ***expression***

Explanation: Assigns the *expression* to the given *name*.

EVEN

Explanation: If necessary, increments the location counter to an even value and generates one NOP instruction (90h).

EXTRN ***name*** **:** ***type*,,,**

Explanation: Defines an external variable, label, or symbol named *name* and whose type is *type*.

name **GROUP** ***seg-name,,,***

Explanation: Associates a group name *name* with one or more segments.

IF ***expression***

Explanation: Grants assembly if the *expression* is nonzero (true).

IF1

Explanation: Grants assembly on pass 1 only.

IF2

Explanation: Grants assembly on pass 2 only.

IFB <*arg*>

Explanation: Grants assembly if the *arg* is blank.

IFDEF ***name***

Explanation: Grants assembly if *name* is a previously defined label, variable, or symbol.

IFDIF ***<arg1>,< arg2>***

Explanation: Grants assembly if the arguments are different.

IFE ***expression***

Explanation: Grants assembly if the *expression* is zero (false).

IFIDN ***<arg1>,<arg2>***

Explanation: Grants assembly if the arguments are identical.

IFNB ***<arg>***

Explanation: Grants assembly if the *arg* is not blank.

IFNDEF ***name***

Explanation: Grants assembly if *name* has not yet been defined.

INCLUDE ***filename***

Explanation: Inserts source code from the source file given by *filename* into the current source file during assembly.

name **LABEL** ***type***

Explanation: Creates a new variable or label by assigning the current location counter value and the given *type* to *name.*

.LALL

Explanation: Lists all statements in a macro.

.LFCOND

Explanation: Restores the listing of conditional blocks.

.LIST

Explanation: Restores listing of statements in the program listing.

NAME ***module-name***

Explanation: Sets the name of the current module to *module-name.*

ORG ***expression***

Explanation: Sets the location counter to *expression.*

%OUT *text*

Explanation: Displays *text* at the user's terminal.

PAGE *length* , *width*

Explanation: Sets the line length and character width of the program listing.

PAGE +

Explanation: Increments section page numbering.

PAGE

Explanation: Generates a page break in the listing.

name* PROC *type

Explanation: Marks the beginning of a procedure definition.

PUBLIC *name,,,*

Explanation: Makes the variable, label, or absolute symbol given by *name* available to all other modules in the program.

.RADIX ***expression***

Explanation: Sets the input radix for numbers in the source file to *expression.*

recordname **RECORD** ***fieldname:width*** **[=** ***exp*****],,,**

Explanation: Defines an record type for an 8- or 16-bit record that contains one or more fields.

.SALL

Explanation: Suppresses listing of all macro expansions.

name **SEGMENT** ***align combine 'class'***

Explanation: Marks the beginning of a program segment named *name* and having segment attributes *align*, *combine*, and *class.*

.SFCOND

Explanation: Suppresses listing of any subsequent conditional blocks whose IF condition is false.

name **STRUC**

Explanation: Marks the beginning of a type definition for a structure.

SUBTTL ***text***

Explanation: Defines the listing subtitle.

.TFCOND

Explanation: Sets the default mode for listing of conditional blocks.

TITLE ***text***

Explanation: Defines the program listing title.

.XALL

Explanation: Lists only those macro statements that generate code or data.

.XCREF ***name,,,***

Explanation: Suppresses the listing of symbols in the cross-reference listing file. (For DOS transfer usage only)

.XLIST

Explanation: Suppresses listing of subsequent source lines to the program listing.

Appendix C. Segment Names For High-Level Languages

This appendix describes the naming conventions used to form assembly language source files that are compatible with object modules produced by the IBM Personal Computer XENIX C language compiler.

High-level language modules have the following four predefined segment types:

TEXT	For program code
DATA	For program data
BSS	For uninitialized space
CONST	For constant data.

Any assembler language source file that you want to assemble and link to a high-level language module must use these segments as described in the following sections.

High-level language modules also have four memory models:

Small	For single code and data segments
Middle	For multiple code segments but a single data segment
Large	For multiple code and data segments
Huge	For multiple code and data segments whose arrays in C exceed 64K bytes.

For huge memory models, refer to *XENIX C Library Guide and Compiler Reference* for the conventions of addressing arrays that exceed 64K bytes. In addition, a C program should be written using the **-S** option to create an assembly program with an **.s** extension to examine segment declaration, array addressing, and pointer incrementing and decrementing.

Assembly language source files to be assembled for a given memory model must use the naming conventions given in the following sections.

TEXT Segments

Syntax

```
name _TEXT    SEGMENT BYTE PUBLIC 'CODE'
              statements
name_TEXT     ENDS
```

A **TEXT** segment defines a module's program code. It contains *statements* that define instructions and data within the segment. A text segment must have the name *name* **_TEXT**, where *name* can be any valid name. For middle, large, and huge module programs, the module's own name is recommended. For small model programs, only **_TEXT** is allowed.

A segment can contain any combination of instructions and data statements. These statements must appear in an order that creates a valid program. All instructions and data addresses in a text segment are relative to the **CS** segment register. Therefore, this statement:

```
assume cs: name_TEXT
```

must appear at the beginning of the segment. This statement ensures that the assembler will associate each label and variable declared in the segment with the **CS** segment register (see "ASSUME Directive" on page 3-9 for more information).

This table defines loading instructions the assembler passes to the linker for **TEXT** segments:

Alignment	BYTE
Combination Type	PUBLIC
Class Name	CODE

Although other segment attributes are available, you should not use them. For a complete description of the attributes, see "SEGMENT and ENDS Directives" on page 3-4.

Small Model Programs

The assembler allows only one text segment. The segment must not exceed 64K bytes. All procedure and statement labels should have the **NEAR** type. For example:

```
_TEXT   segment byte public 'CODE'
assume cs:_TEXT
_main   proc near

.
.
.
_main   endp
_TEXT   ends
        end
```

Middle and Large Model Programs

The assembler allows multiple text segments. However, no segment can be greater than 64K bytes. To distinguish one segment from another, each should have its own name. Because most modules contain only one **TEXT** segment, the module's name is often used as part of the **TEXT** segment's name. All procedure and statement labels should have the **FAR** type, unless they will only be accessed from within the same segment. For example:

```
SAMPLE_TEXT     segment byte public 'CODE'
        assume  cs:SAMPLE_TEXT
_main   proc    far

.
.
.
_main   endp
SAMPLE_TEXT     ends
        end
```

DATA Segments - Near

Syntax

```
_DATA SEGMENT WORD PUBLIC 'DATA'
 statements
_DATA ENDS
```

A near **DATA** segment defines initialized data that is in the segment pointed to by the **DS** segment register when the program starts execution. The segment is NEAR because all data in the segment is accessible without giving an explicit segment value.

All programs have exactly one near **DATA** segment. Only large and huge model programs can have additional **DATA** segments.

A near DATA segment's name must be **_DATA**. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K bytes of data. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the statements:

```
DGROUP       group _DATA
     assume ds:DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the data segment will be associated with the **DS** segment register and **DGROUP** (see "ASSUME Directive" on page 3-9 and "GROUP Directive" on page 3-8 for more information).

This table defines loading instructions the assembler passes to the linker for near **DATA** segments:

Alignment	WORD
Combination Type	PUBLIC
Class Name	DATA

Although other segment attributes are available, you should not use them. For a complete description of the attributes, see "SEGMENT and ENDS Directives" on page 3-4. For example:

```
DGROUP       group   _DATA
     assume ds:DGROUP

_DATA   segment word public 'DATA'
count   dw      0
array   dw      10 dup(1)
string  db      'Type CANCEL then press Enter', 0ah, 0
_DATA   ends
```

DATA Segments - Far

Syntax

```
name _DATA   SEGMENT WORD PUBLIC 'FAR_DATA'
                statements
name _DATA   ENDS
```

A far **DATA** segment defines data or data space that you can access only by specifying an explicit segment value. Only large or huge model programs can have far **DATA** segments.

A far **DATA** segment's name must be *name* **_DATA**, where *name* can be any valid name. The name of the first variable declared in the segment is recommended. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K bytes of data. All data addresses in the segment are relative to the **ES** segment register. When accessing a variable in a far DATA segment, the **ES** register must be set to the appropriate segment value. Also, the segment override operator must be used with the variable's name (see “Segment Override Operator” on page 5-14 for more information).

This table defines loading instructions the assembler passes to the linker for **DATA** segments:

Alignment	WORD
Combination Type	PUBLIC
Class Name	FAR_DATA

Although other segment attributes are available, you must not use them. For a complete description of the attributes, see “SEGMENT and ENDS Directives” on page 3-4. For example:

```
ARRAY_DATA segment word public 'FAR_DATA'
array  dw  0
       dw  1
       dw  2
       dw  4
table  dw  1600 dup(?)
ARRAY_DATA ends
```

BSS Segments

Syntax

```
_BSS   SEGMENT WORD PUBLIC 'BSS'
    statements
_BSS   ENDS
```

A **BSS** segment defines uninitialized data space. A **BSS** segment's name must be **_BSS**. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K bytes. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the statements:

```
DGROUP  group _BSS
        assume ds:DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the **BSS** segment will be associated with the **DS** segment register and **DGROUP** (see "ASSUME Directive" on page 3-9 and "GROUP Directive" on page 3-8).

Note: The group name **DGROUP** must not be defined in more than one **GROUP** directive in a source file. If a source file contains both a **DATA** and **BSS** segment, use this directive:

```
DGROUP  group  _DATA, _BSS
```

This table defines loading instructions the assembler passes to the linker for **BSS** segments:

Alignment	WORD
Combination Type	PUBLIC
Class Name	BSS

Although other segment attributes are available, you should not use them. For a complete description of the attributes, see "SEGMENT and ENDS Directives" on page 3-4. For example:

```
DGROUP          group   _BSS
       assume ds: DGROUP


_BSS   segment word public 'BSS'
count  dw      ?
array  dw      10 dup(?)
string db      30 dup(?)
_BSS   ends
```

CONST Segments

Syntax

```
CONST SEGMENT WORD PUBLIC 'CONST'
    statements
CONST ENDS
```

A **CONST** segment defines constant data that will not change during program execution. You typically use **CONST** segments in large and huge model programs to hold the segment values of far data segments.

The constant segment's name must be **CONST**. The segment can contain any combination of data *statements* defining constants to be used by the program. The segment must not exceed 64K bytes. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the statements must appear at the beginning of the segment:

```
DGROUP      group CONST
       assume ds:DGROUP
```

These statements ensure that each variable declared in the constant segment will be associated with the **DS** segment register and **DGROUP** (see “ASSUME Directive” on page 3-9 and “GROUP Directive” on page 3-8).

Note: The group name **DGROUP** must not be defined in more than one **GROUP** directive in a source file. If a source file contains a **DATA**, **BSS**, and **CONST** segment, use this directive:

```
DGROUP  group  _DATA,_BSS,CONST
```

This table defines loading instructions the assembler passes to the linker for **CONST** segments:

Alignment	WORD
Combination Type	PUBLIC
Class Name	CONST

Although other segment attributes are available, you should not use them. For a complete description of the attributes, see "SEGMENT and ENDS Directives" on page 3-4. For example:

```
DGROUP       group   CONST
     assume ds:DGROUP

CONST         segment word public 'CONST'
seg1          dw      ARRAY_DATA
seg2          dw      MESSAGE_DATA
CONST         ends
```

In this example, the constant segment receives the segment values of two far data segments: **ARRAY_DATA** and **MESSAGE_DATA**. You must define these data segments elsewhere in the module.

Appendix D. Macro Assembler and Linker Error Messages

This appendix lists and explains the error messages that can be generated by the Macro Assembler and the Linker, **ld**.

Macro Assembler Messages

This section lists and explains the messages displayed by the Macro Assembler. The assembler displays a message whenever it encounters an error during processing. It displays a warning message whenever it encounters questionable statement syntax.

An end-of-assembly message is displayed at the end of processing, even if no errors occurred. The message contains a count of errors and warning messages it displayed during the assembly. The message has the form:

```
n   Bytes of symbol space free

n   Warning Errors
n   Severe Errors
```

This message is also copied to the source listing.

Error messages are divided into two categories: assembler errors and I/O handler errors. In each category, messages are listed in numerical order with a short explanation where necessary. All messages are fatal except as noted with an asterisk (*).

An error message will have the following form:

Name	**Line**	**Error #**	**Message**
program.s	(1256) :	error 4:	Redefinition of symbol

where **Name** is the name of the source file, **Line** is the line number of the source file where the error occurred, **Error #** is the error message number, and **Message** is the error message text.

Assembler Errors

Block nesting error

Explanation: Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is closing an outer level of nesting with inner level(s) still open.

*1 **Extra characters on line**

Explanation: This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

2 **Register already defined**

Explanation: This only occurs if the assembler has internal logic errors.

3 **Unknown symbol type**

Explanation: Symbol statement has something in the type field that is unrecognizable.

4 **Redefinition of symbol**

Explanation: This error occurs on pass 2 and succeeding definitions of a symbol.

5 **Symbol is multi-defined**

Explanation: This error occurs on a symbol that is later redefined.

6 **Phase error between passes**

Explanation: The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2, causing the next label to change. You can use the /D option to produce a listing to aid in resolving phase errors between passes.

7 **Already had ELSE clause**

Explanation: Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF . . . ENDIF).

8 **Not in conditional block**

Explanation: An ENDIF or ELSE is specified without a previous conditional assembly directive active.

9 **Symbol not defined**

Explanation: A symbol is used that has no definition.

10 **Syntax error**

Explanation: The syntax of the statement does not match any recognizable syntax.

11 **Type illegal in context**

Explanation: The type specified is of an unacceptable size.

12 **Should have been group name**

Explanation: The assembler was expecting a group name but something other than this was given.

13 **Must be declared in pass 1**

Explanation: An item was referenced before it was defined in Pass 1. For example, `IF DEBUG` is illegal if `DEBUG` is not previously defined.

14 **Symbol type usage illegal**

Explanation: A PUBLIC symbol was used illegally.

15 **Symbol already different kind**

Explanation: An attempt was made to define a symbol differently from a previous definition.

16 **Symbol is reserved word**

Explanation: An assembler reserved word was used illegally (for example, to declare MOV as a variable).

17 **Forward reference is illegal**

Explanation: An attempt was made to reference something before it is defined in pass 1.

18 **Must be register**

Explanation: A register was expected as an operand but you furnished a symbol.

19 **Wrong type of register**

Explanation: A directive or instruction was expected for one type of register, but another was specified (for example, INC CS).

20 **Must be segment or group**

Explanation: The assembler expected a segment or group and something else was specified.

21 **Symbol has no segment**

Explanation: A variable was used with SEG, but the variable has no known segment.

22 **Must be symbol type**

Explanation: Must be WORD, DW, QW, BYTE, or TB but the assembler received something else.

23 **Already defined locally**

Explanation: A symbol was defined as EXTERNAL, but it had already been defined locally.

24 **Segment parameters are changed**

Explanation: The list of arguments to SEGMENT were not identical to the first time this segment was used.

25 **Not proper align/combine type**

Explanation: SEGMENT parameters are incorrect.

26 **Reference to multi-defined**

Explanation: The instruction references something that has been multi-defined.

27 **Operand was expected**

Explanation: The assembler was expecting an operand but an operator was received.

28 **Operator was expected**

Explanation: The assembler was expecting an operator but an operand was received.

29 **Division by 0 or overflow**

Explanation: An expression is given that results in a divide by 0 or a number larger than can be represented.

30 **Shift count is negative**

Explanation: A shift expression is generated that results in a negative shift count.

31 **Operand types must match**

Explanation: The assembler got different kinds or sizes of arguments in a case where they must match (for example, MOV).

32 **Illegal use of external**

Explanation: An external was used in some illegal manner (for example, DB M DUP(?) where M is declared external).

33 **Must be record field name**

Explanation: The assembler expected a record field name but got something else.

34 **Must be record or field name**

Explanation: The assembler expected a record name or field name and received something else.

35 **Operand must have size**

Explanation: The assembler expected an operand to have a size, but it did not.

36 **Must be var, label or constant**

Explanation: The assembler expected a variable, label, or constant but received something else.

37 **Must be structure field name**

Explanation: The assembler expected a structure field name but received something else.

38 **Left operand must have segment**

Explanation: The program used something in the right operand that required a segment in the left operand (for example, :.).

39 **One operand must be const**

Explanation: This is an illegal use of the addition operator.

40 **Operands must be same or 1 abs**

Explanation: This is an illegal use of the subtraction operator.

41 **Normal type operand expected**

Explanation: The assembler received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

42 **Constant was expected**

Explanation: The assembler expected a constant and received an item that does not evaluate to a constant (for example, a variable name or external).

43 **Operand must have segment**

Explanation: The SEG directive was used illegally.

44 **Must be associated with data**

Explanation: A code-related item was used where a data-related item was expected (for example, MOV AX, <code-label>).

45 **Must be associated with code**

Explanation: A data-related item was used where a code-related item was expected.

46 **Already have base register**

Explanation: The program tried to double a base register.

47 **Already have index register**

Explanation: The program tried to double an index address.

48 **Must be index or base register**

Explanation: Instruction requires a base or index register and some other register was specified in square brackets, `[ ]`.

49 **Illegal use of register**

Explanation: A register was used with an instruction where there is no 8086 or 8088 instruction possible.

50 **Value is out of range**

Explanation: Value is too large for expected use (for example, MOV AL,5000).

51 **Operand not in IP segment**

Explanation: Access of operand is impossible because it is not in the current IP segment.

52 **Improper operand type**

Explanation: An operand was used such that the opcode cannot be generated.

53 **Relative jump out of range**

Explanation: Relative jumps must be within the range -128 to +127 of the current instruction, and the specific jump is beyond this range.

54 **Index displ. must be constant**

Explanation: Index display was used illegally.

55 **Illegal register value**

Explanation: The register value specified does not fit into the reg field (the value is greater than 7).

56 **No immediate mode**

Explanation: Immediate mode was specified for an opcode that cannot accept the immediate mode (for example, PUSH).

57 **Illegal size for item**

Explanation: Size of referenced item is illegal (for example, shift of a double word).

58 **Byte register is illegal**

Explanation: One of the byte registers was used in a context where it is illegal (for example, "PUSH AL," is illegal).

59 **CS register illegal usage**

Explanation: An attempt was made to use the CS register illegally. For example, "XCHG CS,AX," is illegal.

60 **Must be AX or AL**

Explanation: Some register was specified other than AX or AL where only these are acceptable (for example, the IN instruction).

61 **Improper use of segment reg**

Explanation: A segment register was specified where this is illegal (for example, an immediate move to a segment register).

62 **No or unreachable CS**

Explanation: The program tried to jump to a label that is unreachable.

63 **Operand combination illegal**

Explanation: A two-operand instruction was specified where the combination specified is illegal.

64 **Near JMP/CALL to different CS**

Explanation: The program attempt to do a NEAR jump or call to a location in a different CS ASSUME.

65 **Label can't have seg. override**

Explanation: Segment override was used illegally.

66 **Must have opcode after prefix**

Explanation: A REPE, REPNE, REPZ, or REPNZ instruction was used without specifying any opcode after it.

67 Can't override ES segment

Explanation: The program tried to override the ES segment in an instruction where this override is not legal. For example, "STOS DS:TARGET" is illegal.

68 Can't reach with segment reg

Explanation: There is no ASSUME that makes the variable reachable.

69 Must be in segment block

Explanation: The program attempted to generate code when not in a segment.

70 Can't use EVEN on BYTE segment

Explanation: A segment was declared to be a byte segment, and an attempt to use EVEN was made.

72 Illegal value for DUP count

Explanation: DUP counts must be a constant that is not 0 or negative.

73 Symbol already external

Explanation: The program attempted to define a symbol as local that is already external.

74 DUP is too large for linker

Explanation: Nesting of DUPs was such that too large a record was created for the linker.

75 **Usage of ? (indeterminate) bad**

Explanation: The ? was used illegally (for example, ?+5).

76 **More values than defined with**

Explanation: Too many initial values were given when defining a variable using a REC or STRUC type.

77 **Only initialize list legal**

Explanation: The program attempted to use a STRUC name without angle brackets, < >.

78 **Directive illegal in STRUC**

Explanation: All statements within STRUC blocks must either be comments preceded by a semicolon (;) or one of the define directives.

79 **Override with DUP is illegal**

Explanation: In a STRUC initialization statement, you tried to use DUP in an override.

80 **Field cannot be overridden**

Explanation: In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

81 **Override is of wrong type**

Explanation: In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

83 **Circular chain of EQU aliases**

Explanation: An alias EQU eventually points to itself.

84 **8087 opcode can't be emulated**

Explanation: Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

*85 **End of file, no END directive**

Explanation: You forgot an end statement or there is a nesting error.

86 **Data omitted with no segment**

Explanation: The data was defined outside of a SEGMENT - ENDS pair.

98 **Override value is wrong length**

Explanation: There is an improper sized value in a RECORD or STRUC field.

Linker Messages

This section lists the error messages that can occur when linking programs. The messages are in alphabetical order.

-A and -F are mutually exclusive

Explanation: The -A and -F switches are mutually exclusive.

Address missing

Explanation: The -A switch is missing a following number.

Array element size mismatch

Explanation: A far communal array has been declared with two or more different array element sizes (for example, declared once as an array of characters and once as an array of reals).

Note: Communal arrays are not available in the assembler.

Attempt to access data outside segment bounds

Explanation: An **LEDATA** or an **LIDATA** record specifies bytes that occupy offsets beyond the end of the segment as defined by the corresponding **SEGDEF** record.

Attempt to put segment name in more than one group in file *filename*

Explanation: A segment is declared to be a member of two different groups. Correct the source and recreate the object files.

Cannot create list file

Explanation: The linker was unable to create the list (map) file. Possibly the user does not have permission, or the disk is full.

Cannot find file

Explanation: You specified an object module or library file that the linker cannot find or is not able to open for reading.

Cannot open run file

Explanation: The directory or disk is full, or you do not have the right permissions. Make space on the disk or in the directory, or change permissions.

Cannot open temporary file

Explanation: The directory or disk is full. Make space on the disk or in the directory.

Common area longer than 65536 bytes

Explanation: Your program has more than 64K bytes of communal variables.

Note: Only C programs can cause this message to be displayed. Rewrite your program using fewer communal variables or making some of your communal variables **far**; or recompile your program as a large or huge model.

Data record too large

Explanation: LEDATA record (in an object module) contains more than 1024 bytes of data. This is a translator error. Either the C compiler or assembler produced an incorrect object module.

Dup record too large

Explanation: LIDATA record (in an object module) contains more than 512 bytes of data. Most likely, an assembly module contains a struc definition that is very complex, or a series of deeply nested DUP statements (for example, ARRAY db 10 dup(11 dup (12 dup (13 dup (. . .))))). Simplify and reassemble.

Error accessing library

Explanation: The linker was unable to open a specified library. Make sure the file exists and has the proper permissions.

Fixup overflow near num in segment name in filename(name) offset num

Explanation: Some possible causes are: (1) A group is larger than 64K bytes, (2) the user's program contains an intersegment short jump or intersegment short call, (3) the user has a data item whose name conflicts with that of a subroutine in a library included in the link, and (4) the user has an EXTRN declaration inside the body of a segment, for example:

```
CODE    segment public 'code'
extrn   main:far
start   proc    far
        call    main
        ret
start   endp
CODE    ends
```

The following construction is preferred:

```
extrn   main:far
CODE    segment public 'code'
start   proc    far
        call    main
        ret
start   endp
CODE    ends
```

Revise the source and recreate the object file.

Group *name* larger than 64K bytes

Explanation: A group *name* has been defined that is larger than the maximum allowed for a physical segment. Reduce the size of the group.

Invalid object module

Explanation: One of the object modules is invalid. Try recompiling.

List file name missing

Explanation: The **-m** switch is missing a following string.

Multiple code segments--should be medium model

Explanation: There is more than one code segment and the **-Mm**, **-Ml**, or **-Me** switch was not given. Make sure all modules have the same memory model or link with the **-Me** switch. Not fatal.

Multiple data segments--should be large model

Explanation: There is more than one data segment and the **-Ml** or **-Me** switch was not given. Make sure all modules have the same memory model or link with the **-Me** switch. Not fatal.

Name length missing

Explanation: The **-n** switch is missing a following number.

NEAR/HUGE conflict

Explanation: Near and huge definitions for a communal variable conflict.

Note: Communal variables are not available in the assembler.

No scratch file

Explanation: Internal or system error. Do a file system check.

Number missing

Explanation: The **-S** switch is missing a following number.

Object not found

Explanation: You are specifying an object module or library file that the linker cannot find or is not able to open for reading.

Out of space on list file

Explanation: Disk on which list file is being written is full. Free more space on the disk and try again.

Out of space on run file

Explanation: Disk on which executable file is being written is full. Free more space on the disk and try again.

Run file name missing

Explanation: The **-o** switch is missing a following string.

Stack size missing

Explanation: The **-F** switch is missing a following number.

Segment limit too high

Explanation: There is insufficient memory for the link editor to allocate tables to describe the number of segments requested (either the value specified with **-S** or the default: 128). Try the link again, using the **-S** option to select a smaller number of segments (for example, 64 if the default was used previously), or free some memory.

Segment size exceeds 64K

Explanation: You have a small model program with more than 64K bytes of code, or you have a middle model program with more than 64K bytes of data. Try compiling and linking a middle, large, or huge model (as appropriate).

Stack size missing

Explanation: The **-F** switch lacks a following hexadecimal number.

Symbol already defined: <*symbol*>

Explanation: A public symbol has been defined more than once.

Symbol missing

Explanation: The **-u** switch lacks a following string.

Symbol table overflow

Explanation: Your program has more than 256K bytes of symbolic information (publics, externs, groups, classes, file, etc.). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

<*switch*> ignored

Explanation: You used an unrecognized switch.

Terminated by user

Explanation: You pressed an interrupt key (**Del**), which caused the link editor to terminate.

Too many external symbols in one module

Explanation: Your object module specified more than the allowed number of external symbols. Break up the module.

Too many group-, segment-, and class-names in one module

Explanation: Your program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes and recreate the object files.

Too many groups

Explanation: Your program defines more than nine groups. Reduce the number of groups.

Too many GRDEFs in one module

Explanation: Link editor encountered more than nine **GRDEF**s in a single module. Reduce the number of **GRDEF**s or split up the module.

Too many libraries

Explanation: You tried to link with more than 16 libraries. Combine libraries or link modules that require fewer libraries.

Too many segments

Explanation: Your program has too many segments. Relink using the **-S** option with an appropriate number of segments specified.

Too many segments in one module

Explanation: Your object module has more than 255 segments. Split the modules or combine segments.

Too many TYPDEFs

Explanation: TYPDEFs are records emitted by the compiler to describe communal variables. Create two sources from the old source, dividing the communal variable definitions between them; re-compile and re-link.

Unexpected end-of-file on scratch file

Explanation: The temporary scratch file has probably been removed.

-u seen before -n <*num*>

Explanation: An undefined symbol has been given before the maximum name-length switch. Reverse the order of the two switches.

unknown model specifier <-M?>

Explanation: The **-M** switch was given with a character following that was not equal to **s**, **m**, **l**, **h**, or **e**.

Unrecognized PC XENIX version number

Explanation: The version number following the **-v** switch must currently be either 2 or 3.

Use -i switch

Explanation: There is more than one segment and the program is being linked impure, that is without the **-i** switch. Impure executables can only have one segment.

Version number missing

Explanation: You gave the **-v** switch without a version number following.

WARNING: Groups *name1* and *name2* overlap

Explanation: Input to the linker has established a segment ordering such that the first segment of one group is ordered before the last segment of the previous group. Change the segment ordering or redefine appropriate class names.

WARNING: model mismatch

Explanation: You linked object modules with different memory models. Recompile so all modules have the same model, or else use the **-Me** switch.

WARNING: __.SYMDEF out of date in ***< library name >***

Explanation: A library archive (**.a**) file has been modified since the last time its dictionary was updated with **ranlib**. Re-**ranlib** the file. Users who install new libraries with the **cp** command may see this error message when in fact the library is up-to-date. This is because **cp** makes the modification time of the new library equal to the time the library was copied and the dictionary time is left unchanged. To avoid this situation, always copy libraries with copy **-m** or extract them from **tar**-format archive files.

WARNING: too many public symbols

Explanation: You asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The link editor will produce an unsorted listing of the public symbols.

Index

Special Characters

.CREF directive 9-8
.LALL directive 9-8
.LFCOND directive 9-7
.LIST directive 9-6
.RADIX directive 9-2
.SALL directive 9-8
.SFCOND directive 9-7
.TFCOND directive 9-7
.XALL directive 9-8
.XCREF directive 9-8
.XLIST directive 9-6
.TYPE operator 5-17
.186 directive 3-4
.286c directive 3-4
.286p directive 3-4
.287 directive 3-4
.8086 directive 3-3
.8087 3-4
&, substitute operator 8-10
%OUT directive 9-3
: segment override operator 5-14
= directive 4-9

A

ABS type numbers. 6-3
absolute segments 3-6
alignment 3-10
  segments 3-6
alphabetical ordering 1-7
AND operator 5-12
arithmetic operators 5-10
as
  converting to uppercase 1-9
  define symbol 1-7
  description iii
  options 1-6
  output error messages 1-12
  output object code 1-10
  print symbol information 1-10
  produce listing file 1-8
  starting 1-1
  using 1-1
assembler, macro iii
assembly
  false conditionals 9-7
  macros 9-8
  page breaks 9-5
  page dimensions 9-5
  subtitle 9-5
  suppressing 9-6
  symbols 9-8
  title 9-4

assembly listing, reading 1-13
assembly statistics 1-11
ASSUME directive 3-9
AT segments 3-6

B

based indexed operands 5-6
based operands 5-5
bitwise operators 5-12
boldface iv
brackets iv
BSS C-1

C

calls, macro 8-4
character constant 2-4
character set 2-1
character, literal 8-12
classes 3-7
CMPS instruction A-8
code, program 1-13
combine type 3-6
COMMENT directive 2-8
comment, macro 8-13
comments 2-8
COMMON segments 3-6
conditional directives
  assembly passes 7-2
  directives 7-1
  expressions 7-2
  macro arguments 7-3, 7-4
  nesting 7-1
  symbols 7-3
conditionals, false 1-12
CONST C-1
CONST segments C-7
constant
  default radix 9-2
constant operands 5-2
constants
  character 2-4
  string 2-4
conventions
  coding 1-1
  notational iv
cross-reference listing symbols 9-9
CX register A-8

D

data C-1
  bytes 4-3
  doublewords 4-5
  quadwords 4-6
  ten-byte words 4-7
  words 4-4
data declarations 4-2
DATA segments
  far C-5
  near C-3
DB directive 4-3
DD directive 4-5
declarations
  byte data 4-3
  data 4-2
  doubleword data 4-5
  label 4-1
  quadword data 4-6
  record 4-14, 4-15
  structure 4-14
  symbol 4-8
  type 4-11
  word data 4-4
  10-byte words 4-7

default segment
 registers 3-9
direct memory
 operands 5-2
directives B-1
  .CREF 9-8
  .LALL 9-8
  .LFCOND 9-7
  .LIST 9-6
  .RADIX 9-2
  .SALL 9-8
  .SFCOND 9-7
  .TFCOND 9-7
  .XALL 9-8
  .XCREF 9-8
  .XLIST 9-6
  .RADIX 2-2
  .186 3-4
  .286c 3-3, 3-4
  .286p 3-4
  .287 3-3, 3-4
  .8086 3-3
  .8087 3-4
  %OUT 9-3
  = 4-9
  ASSUME 3-9
  COMMENT 2-8
  conditional 7-1
  data declaration 4-2
  DB 4-3
  DD 4-5
  DQ 4-6
  DT 2-4, 4-7
  DW 4-4
  END 3-7
  ENDM 8-2, 8-7, 8-9
  ENDP 3-11
  ENDS 3-4, 4-11
  EQU 4-9
  EVEN 3-10
  EXITM 8-9
  EXTRN 6-2
  global declaration 6-1
  GROUP 3-8
  IF 7-2
  IFB 7-3
  IFDEF 7-3
  IFDIF 7-4
  IFE 7-2
  IFIDN 7-4
  IFNB 7-3
  IFNDEF 7-3
  IF1 7-2
  IF2 7-2
  INCLUDE 9-2
  instruction set 3-3
  IRP 8-7
  IRPC 8-9
  LABEL 4-10
  LOCAL 8-5
  MACRO 8-2
  mnemonic 2-7
  mnemonics 2-6
  modular
   programming 6-1
  NAME 9-4
  ORG 3-10
  PAGE 9-5
  PROC 4-2
  program structure 3-1
  PUBLIC 6-2
  PURGE 8-6
  RECORD 4-13, 4-15
  REPT 8-7
  SEGMENT 3-4
  STRUC 4-11
  SUBTITLE 9-5
  TITLE 9-4
  type declarations 4-11
DQ directive 4-6
DT directive 2-4, 4-7
DUP operator 4-8
DW directive 4-4

E

emulator, floating-point 1-11
encoded real numbers 2-3
END directive 3-7
ENDM directive
  IRP directive 8-7
  IRPC directive 8-9
  MACRO directive 8-2
  REPT directive 8-7
ENDP directive 3-11
ENDS directive
  SEGMENT directives 3-4
  STRUC directive 4-11
entry point 3-7
EQ operator 5-11
EQU directive 4-9
error messages, link editor D-15
EVEN directive 3-10
EXITM directive 8-9
expression operator 8-12
expressions 5-9
external symbols 6-2
EXTRN directive 6-2

F

false conditionals, list 1-12
files, source 3-1
flag register 5-4
floating-point emulator 1-11
floating-point processor 1-11
forward references 5-21

G

GE operator 5-11
generate reference listings 9-9
global symbols 6-1
group
  definition 3-8
  size restriction 3-8
GROUP directive 3-8
group table 1-17
GT operator 5-11

H

hexadecimal numbers 2-2
HIGH operator 5-16

I

IF directive 7-2
IFB directive 7-3
IFDEF directive 7-3
IFDIF directive 7-4
IFE directive 7-2
IFIDN directive 7-4
IFNB directive 7-3
IFNDEF directive 7-3
IF1 directive 7-2
IF2 directive 7-2
INCLUDE directive 9-2
index operator 5-13
indexed operands 5-5
instruction mnemonic 2-7
instruction mnemonics 2-6

instruction set directives 3-3
instructions
  80186 A-13
  80286 nonprotected A-14
  80286 protected A-15
  80287 A-16
  8086 A-2
  8087 A-9
integer, default radix 9-2
IRP directive 8-7
IRPC directive 8-9
italics iv

K

keyword, NOTHING 3-10

L

LABEL directive 4-10
labels
  declarations 4-1
  default segments 3-9
  near 4-1
  procedure 4-2
  procedures 3-11
language modules C-1
LE operator 5-11
LENGTH operator 5-18
link editor D-15
listing false conditionals 1-12
listing, pass 1
  creating 1-7
  reading 1-20
listing, read 1-13
literal character operator 8-12
literal text operator 8-11
loading
  segments 3-6
loading options 3-6
LOCAL directive 8-5
location counter 5-3
LODS instruction A-8
LOW operator 5-16
lowercase
  external names, in 1-9
  names, preserving 1-9
  public names, in 1-9
LT operator 5-11

M

macro assembler iii
macro calls 8-4
macro comment 8-13
MACRO directive 8-2
macro library 8-6
macro table 1-15
macros
  argument testing 7-4
MASK operator 5-20
memory image
  loading options 3-6
MEMORY segments 3-6
message
  user console 9-3
mnemonics 2-6
module
  end 3-7
  main 3-7
  name 9-4
MOVS instruction A-8

N

NAME directive 9-4
names 2-5
  groups 3-8
  module 9-4
  register 2-6
  reserved 2-6
  segment classes 3-7
  segments 3-4
names, lowercase 1-9
NE operator 5-11
near label
 declarations 4-1
nested conditionals 7-1
nonprotected instructions,
 80286 A-14
NOT operator 5-12
notational conventions iv
NOTHING keyword 3-10
numbers
  encoded real 2-3
  packed decimal 2-4
  real 2-3

O

OFFSET operator 5-16
operand data types iii
operands 5-9
  based 5-5
  based indexed 5-6
  constant 5-2
  direct memory 5-2
  indexed 5-5
  location counter 5-3
  record 5-8
  record field 5-8
  register 5-3
  relocatable 5-2
  strong typing 5-23
  structure 5-7
operator
  expression 8-12
  literal character 8-12
  literal text 8-11
  substitute 8-10
operator, DUP 4-8
operators 5-9
  .TYPE 5-17
  arithmetic 5-10
  bitwise 5-12
  HIGH 5-16
  index 5-13
  LENGTH 5-18
  LOW 5-16
  MASK 5-20
  OFFSET 5-16
  PTR 5-13
  relational 5-11
  SEG 5-16
  segment override
   (:) 5-14
  shift 5-11
  SHORT 5-15
  SIZE 5-19
  THIS 5-15
  TYPE 5-17
  WIDTH 5-19
options
  -a 1-7
  -d 1-7
  -I 1-8
  -l 1-8
  -Ml 1-9
  -Mu 1-9
  -Mx 1-9
  -n 1-10
  -o 1-10
  -r 1-11
  -v 1-11
  -x 1-12
OR operator 5-12

ORG directive 3-10
organization for program 1-1
output file
  user console 9-3
output, segments 1-7

P

packed decimal numbers 2-4
PAGE directive 9-5
pass 1 listing
  creating 1-7
  reading 1-20
path names, include 1-8
precedence of operators 5-20
private segments 3-7
PROC directive
  creating labels 4-2
  using 3-11
procedure labels 4-2
procedures 3-11
processor, floating-point 1-11
program
  entry point 3-7
  loading options 3-6
  segments 3-4
program code, reading 1-13
program organization 1-1
program structure directives 3-1
PTR operator 5-13
PUBLIC directive 6-2
PUBLIC segments 3-6
public symbols 6-2
PURGE directive 8-6

R

real numbers
record declarations 4-15
RECORD directive 4-13
record field operand 5-8
record operands 5-8
record table 1-15
reference, cross 9-9
register operands 5-3
relational operators 5-11
relocatable operands 5-2
REP instruction A-8
REPE instruction A-8
repeat instruction A-8
REPNE instruction A-8
REPNZ instruction A-8
REPT directive 8-7
REPZ instruction A-8
reserved names 2-6

S

SCAS instruction A-8
SEG operator 5-16
segment
  alignment 3-6
  AT 3-6
  classes 3-7
  combine types 3-6
  COMMON 3-6
  definition 3-4
  directive 3-4
  groups 3-8
  loading options 3-6
  MEMORY 3-6
  nesting 3-5
  origin 3-10

  - private 3-7
  - PUBLIC 3-6
  - STACK 3-6
- SEGMENT directive 3-4
- segment override (:) operator 5-14
- segment table 1-17
- segments
  - CONST C-7
  - DATA, far C-5
  - DATA, near C-3
  - TEXT C-2
- segments, output 1-7
- shift operators 5-11
- SHL operator 5-11
- SHORT operator 5-15
- SHR operator 5-11
- SIZE operator 5-19
- source file
  - definition 3-1
  - ending 3-7
  - including 9-2
- STACK segments 3-6
- statements 2-7
- statistics, assembly 1-11
- STOS instruction A-8
- string constant 2-4
- string instruction
  - CMPS A-8
  - LODS A-8
  - MOVS A-8
  - SCAS A-8
  - STOS A-8
- strong typing 5-23
- STRUC directives 4-11
- structure declarations 4-14
- structure operands 5-7
- structure table 1-15
- style conventions iv
- substitute operator 8-10
- SUBTITLE
  - assembly listing 9-5
- SUBTTL Directive 9-5
- symbol declarations 4-8
- symbol table 1-18
- symbols
  - absolute 4-9
  - aliases 4-9
  - default segments 3-9
  - external 6-2
  - global 6-1
  - labels 4-10
  - public 6-2
  - relocatable operands 5-2
  - variables 4-10

T

- table
  - group 1-17
  - record 1-15
  - segment 1-17
  - structure 1-15
  - symbol 1-18
- table, macro 1-15
- TEXT C-1
- TEXT segments C-2
- text, literal 8-11
- THIS operator 5-15
- TITLE directive 9-4
- title, assembly listing 9-4
- type declarations 4-11
- TYPE operator 5-17
- types
  - operand matching 5-23
  - record 4-13
  - structure 4-11

V

variables, default
  segments 3-9

W

WIDTH operator 5-19

X

XOR operator 5-12

Numerics

80186 instructions A-13
80286 instructions
  nonprotected A-14
  protected A-15
80287 instructions A-16
8086 instructions A-2
8087 instructions A-9

The Personal
Computer
Programming
Family

Reader's Comment Form

XENIX Operating System Macro Assembler

SV21-8079

Your comments assist us in improving our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for questions regarding setup, operation, or program support or for requests for additional publications. Instead, contact your authorized IBM Personal Computer dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
DEPARTMENT 997
11400 BURNET ROAD
AUSTIN, TEXAS 78758

Fold here

Tape Do not staple Tape

IBM

The Personal
Computer
Programming
Family

Reader's Comment Form

XENIX Operating System Macro Assembler

SV21-8079

Your comments assist us in improving our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for questions regarding setup, operation, or program support or for requests for additional publications. Instead, contact your authorized IBM Personal Computer dealer in your area.

Comments:

Tape

Do not staple

Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
DEPARTMENT 997
11400 BURNET ROAD
AUSTIN, TEXAS 78758

© IBM Corporation 1986
All rights reserved.

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

59X8654